O'REILLY®

TURING 图灵程序设计丛书



Python机器学习基础教程

Introduction to Machine Learning with Python

以机器学习算法实践为重点，使用scikit-learn库从头构建机器学习应用

[德] Andreas C. Müller [美] Sarah Guido 著

张亮（hysic）译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

译者介绍

张亮（hysic）

毕业于北京大学物理学院，爱好机器学习和数据分析的核安全工程师。

图灵程序设计丛书

Python机器学习基础教程

Introduction to Machine Learning with Python

［德］Andreas C. Müller ［美］Sarah Guido 著
张亮（hysic） 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Python机器学习基础教程 / (德) 安德里亚斯·穆勒 (Andreas C. Müller), (美) 莎拉·吉多 (Sarah Guido) 著 ; 张亮 (hysic) 译. -- 北京 : 人民邮电出版社, 2018.1
（图灵程序设计丛书）
ISBN 978-7-115-47561-9

Ⅰ. ①P… Ⅱ. ①安… ②莎… ③张… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2017)第314601号

内 容 提 要

本书是机器学习入门书，以 Python 语言介绍。主要内容包括：机器学习的基本概念及其应用；实践中最常用的机器学习算法以及这些算法的优缺点；在机器学习中待处理数据的呈现方式的重要性，以及应重点关注数据的哪些方面；模型评估和调参的高级方法，重点讲解交叉验证和网格搜索；管道的概念；如何将前面各章的方法应用到文本数据上，还介绍了一些文本特有的处理方法。

本书适合机器学习从业者或有志成为机器学习从业者的人阅读。

◆ 著　　[德] Andreas C. Müller [美] Sarah Guido
译　　张 亮（hysic）
责任编辑　岳新欣
执行编辑　李 敏
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：18.75
字数：443千字　　2018年1月第1版
印数：1－4 000册　　2018年1月北京第1次印刷

著作权合同登记号　图字：01-2017-8626号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

目前，从医疗诊断和治疗到在社交网络上寻找好友，许多商业应用和研究项目都离不开机器学习。许多人以为，只有大公司的大型研究团队才能用到机器学习。在本书中，我们要向你展示，自己动手构建机器学习解决方案是多么容易的一件事，也将介绍如何将这件事做到最好。学完本书中的知识，你可以自己构建系统，研究 Twitter 用户的情感，或者对全球变暖做出预测。机器学习的应用十分广泛，如今的海量数据使得其应用范围更是远超人们的想象。

目标读者

本书是为机器学习从业者或有志成为机器学习从业者的人准备的，他们在为现实生活中的机器学习问题寻找解决方案。这是一本入门书，不需要读者具备机器学习或人工智能（artificial intelligence，AI）的相关知识。我们主要使用 Python 和 `scikit-learn` 库，一步一步构建一个有效的机器学习应用。我们介绍的方法适用于科学家和研究人员，也会对开发商业应用的数据科学家有所帮助。如果你对 Python 以及 NumPy 和 `matplotlib` 库有所了解的话，将能够更好地掌握本书的内容。

我们刻意不将数学作为重点，而是将机器学习算法的实践作为重点。数学（尤其是概率论）是机器学习算法的基石，所以我们不会详细分析算法的细节。如果你对机器学习算法的数学部分感兴趣，我们推荐阅读 Trevor Hastie、Robert Tibshirani 和 Jerome Friedman 合著的《统计学习基础》（*Elements of Statistical Learning*，Springer 出版社）一书，可以在几位作者的网站上免费阅读这本书（http://statweb.stanford.edu/~tibs/ElemStatLearn/）。我们也不会从头讲解如何编写机器学习算法，而是将重点放在如何应用 `scikit-learn` 库和其他库中已经实现的海量模型。

写作本书的原因

市面上已经有许多关于机器学习和 AI 的书了，但这些书都是为计算机科学专业的研究生或博士生准备的，里面全都是高等数学的内容。与之形成鲜明对比的是，在研究领域和商业应用中，机器学习是作为一般工具使用的。如今，使用机器学习并不需要拥有博士学

位。然而，能够完全涵盖在实践中实现机器学习算法的所有重要内容，而又不需要先修高等数学课程，这样的学习资源少之又少。对于那些想要使用机器学习算法而又不想花费大量时间研读微积分、线性代数和概率论的人来说，我们希望本书能够有所帮助。

本书概览

本书的结构大致如下。

- 第 1 章介绍机器学习的基本概念及其应用，并给出本书会用到的基本设置。
- 第 2 章和第 3 章介绍实践中最常用的机器学习算法，并讨论这些算法的优缺点。
- 第 4 章介绍在机器学习中待处理数据的呈现方式的重要性，以及应重点关注数据的哪些方面。
- 第 5 章介绍模型评估和调参的高级方法，重点讲解交叉验证和网格搜索。
- 第 6 章解释管道的概念。管道用于串联多个模型并封装工作流。
- 第 7 章介绍如何将前面各章讲述的方法应用到文本数据上，还介绍了一些文本特有的处理方法。
- 第 8 章对全书进行总结，还介绍了有关更高级主题的参考资料。

虽然第 2 章和第 3 章给出了实际算法，但对于初学者来说，并不需要理解所有这些算法。如果你想要尽快构建一个机器学习系统，我们建议你首先阅读第 1 章和第 2 章的开始部分，里面介绍了所有的核心概念。然后你可以翻到 2.5 节，里面提到了我们介绍的所有监督学习模型。从中选择最适合你需求的模型，然后翻回到对应小节阅读其详细内容。之后你可以使用第 5 章中的方法对你的模型进行评估和调参。

在线资源

在学习本书时，一定要参考 `scikit-learn` 官方网站（http://scikit-learn.org），查阅关于类和函数的更详细的文档，以及很多示例。此外，Andreas Müller 创建的视频课程“scikit-learn 高等机器学习”（Advanced Machine Learning with scikit-learn）可以作为本书的补充材料。你可以在 http://shop.oreilly.com/product/0636920043836.do 观看该课程。

排版约定

本书使用了下列排版约定。

- **黑体**
  表示新术语或重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。也用于表示命令、模块和包的名称。
- 加粗等宽字体（**`constant width bold`**）
  表示需要用户逐字输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）
  表示应替换成用户输入的值，或替换成根据上下文确定的值。



该图标表示提示或建议。



该图标表示一般性说明。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、IPython notebook 等）可以在 https://github.com/amueller/introduction_to_ml_with_python 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序和文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可。销售或分发 O'Reilly 图书的示例光盘则需要获得许可。引用本书中的示例代码来回答问题无需获得许可。将书中大量示例代码放到你的产品文档中则需要获得许可。

如果你在引用本书内容时注明出处，我们将不胜感激，但这并非强制要求。引用说明一般包括书名、作者、出版社和 ISBN。比如："*An Introduction to Machine Learning with Python* by Andreas C. Müller and Sarah Guido (O'Reilly). Copyright 2017 Sarah Guido and Andreas Müller, 978-1-449-36941-5."

如果你认为自己对代码示例的用法超出了合理使用的范围或上述许可的范围，敬请通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online 是应运而生的数字图书馆，它同时以图书和视频的形式出版世界顶级技术和商业作家的专业作品。

技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于企业、政府、教育机构和个人，Safari Books Online 都提供各种产品组合和灵活的定价策略。

用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等数百家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，请访问我们的网站。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们为本书创建了一个网页，在上面列出了本书的勘误表、示例以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920030515.do。

如果你想就本书发表评论或技术性问题，请发送电子邮件到 bookquestions@oreilly.com。

想了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问我们的网站：
http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

来自Andreas的致谢

如果没有许多人的帮助和支持，本书永远不会出版。

我要感谢本书编辑 Meghan Blanchette 和 Brian MacDonald，特别是 Dawn Schanafelt，感谢

他们帮助我和 Sarah 共同完成这本书。

我要感谢我的审稿人 Thomas Caswell、Olivier Grisel、Stefan van der Walt 和 John Myles White，感谢他们花费时间阅读本书的早期版本，并提供宝贵的反馈意见。这些意见也成为了科学计算开源生态系统的基石。

我永远感谢热情的 Python 科学计算开源社区，特别要感谢 `scikit-learn` 的贡献者们。如果没有这个社区的支持和帮助，特别是 Gael Varoquaux、Alex Gramfort 和 Olivier Grisel 的支持和帮助，我永远无法成为 `scikit-learn` 的核心贡献者，也无法像现在这样对这个包有如此深刻的理解。我还要感谢 `scikit-learn` 的其他所有贡献者，他们花费了大量时间改进并维护这个包。

我还要感谢与我讨论的许多同事和同行。这些谈话帮助我理解了机器学习的挑战，并让我产生构思一本教科书的想法。我与许多人讨论过机器学习，但我要特别感谢其中的 Brian McFee、Daniela Huttenkoppen、Joel Nothman、Gilles Louppe、Hugo Bowne-Anderson、Sven Kreis、Alice Zheng、Kyunghyun Cho、Pablo Baberas 和 Dan Cervone。

我还要感谢 Rachel Rakov，她对本书的早期版本做了许多热心的测试和校对工作，在成书过程中给了我很多帮助。

就个人来说，我要感谢我的父母 Harald 和 Margot，还有我的姐姐 Miriam，感谢他们持续给予我的支持和鼓励。我还要感谢生命中的许多人，他们的爱和友谊给我能量，支持我完成这项富有挑战性的任务。

来自Sarah的致谢

我要感谢 Meghan Blanchette，没有她的帮助和指导，甚至就不会有本项目的存在。感谢 Celia La 和 Brian Carlson 早期对本书的审阅。感谢 O’Reilly 工作人员无尽的耐心。最后，感谢 DTS，感谢你永恒不变的支持。

电子书

扫描如下二维码，即可购买本书电子版。

第 1 章
引言

机器学习（machine learning）是从数据中提取知识。它是统计学、人工智能和计算机科学交叉的研究领域，也被称为预测分析（predictive analytics）或统计学习（statistical learning）。近年来，机器学习方法已经应用到日常生活的方方面面。从自动推荐看什么电影、点什么食物、买什么商品，到个性化的在线电台和从照片中识别好友，许多现代化网站和设备的核心都是机器学习算法。当你访问像 Facebook、Amazon 或 Netflix 这样的复杂网站时，很可能网站的每一部分都包含多种机器学习模型。

除了商业应用之外，机器学习也对当前数据驱动的研究方法产生了很大影响。本书中介绍的工具均已应用在各种科学问题上，比如研究恒星、寻找遥远的行星、发现新粒子、分析 DNA 序列，以及提供个性化的癌症治疗方案。

不过，如果想受益于机器学习算法，你的应用无需像上面那些例子那样给世界带来重大改变，数据量也用不着那么大。本章将解释机器学习如此流行的原因，并探讨机器学习可以解决哪些类型的问题。然后将向你展示如何构建第一个机器学习模型，同时介绍一些重要的概念。

1.1　为何选择机器学习

在“智能”应用的早期，许多系统使用人为制订的“if”和“else”决策规则来处理数据，或根据用户输入的内容进行调整。想象有一个垃圾邮件过滤器，其任务是酌情将收到的某些邮件移动到垃圾邮件文件夹。你可以创建一个关键词黑名单，所有包含这些关键词的邮件都会被标记为垃圾邮件。这是用专家设计的规则体系来设计“智能”应用的一个示例。人为制订的决策规则对某些应用来说是可行的，特别是人们对其模型处理过程非常熟悉的应用。但是，人为制订决策规则主要有两个缺点。

- 做决策所需要的逻辑只适用于单一领域和单项任务。任务哪怕稍有变化，都可能需要重写整个系统。
- 想要制订规则，需要对人类专家的决策过程有很深刻的理解。

这种人为制订规则的方法并不适用的一个例子就是图像中的人脸检测。如今，每台智能手机都能够检测到图像中的人脸。但直到 2001 年，人脸检测问题才得到解决。其主要问题在于，计算机“感知”像素（像素组成了计算机中的图像）的方式与人类感知面部的方式有非常大的不同。正是由于这种表征差异，人类想要制订出一套好的规则来描述数字图像中的人脸构成，基本上是不可能的。

但有了机器学习算法，仅向程序输入海量人脸图像，就足以让算法确定识别人脸需要哪些特征。

1.1.1　机器学习能够解决的问题

最成功的机器学习算法是能够将决策过程自动化的那些算法，这些决策过程是从已知示例中泛化得出的。在这种叫作**监督学习**（supervised learning）的方法中，用户将成对的输入和预期输出提供给算法，算法会找到一种方法，根据给定输入给出预期输出。尤其是在没有人类帮助的情况下，给定前所未见的输入，算法也能够给出相应的输出。回到前面垃圾邮件分类的例子，利用机器学习算法，用户为算法提供大量电子邮件（作为输入），以及这些邮件是否为垃圾邮件的信息（作为预期输出）。给定一封新邮件，算法就能够预测它是否为垃圾邮件。

从输入 / 输出对中进行学习的机器学习算法叫作**监督学习算法**（supervised learning algorithm），因为每个用于算法学习的样例都对应一个预期输出，好像有一个“老师”在监督着算法。虽然创建一个包含输入和输出的数据集往往费时又费力，但监督学习算法很好理解，其性能也易于测量。如果你的应用可以表示成一个监督学习问题，并且你能够创建包含预期输出的数据集，那么机器学习很可能可以解决你的问题。

监督机器学习任务的示例如下。

识别信封上手写的邮政编码

这里的输入是扫描的手写数字，预期输出是邮政编码中的实际数字。想要创建用于构建机器学习模型的数据集，你需要收集许多信封。然后你可以自己阅读邮政编码，将数字保存为预期输出。

基于医学影像判断肿瘤是否为良性

这里的输入是影像，输出是肿瘤是否为良性。想要创建用于构建模型的数据集，你需要一个医学影像数据库。你还需要咨询专家的意见，因此医生需要查看所有影像，然后判断哪些肿瘤是良性的，哪些不是良性的。除了影像内容之外，甚至可能还需要做额外的诊断来判断影像中的肿瘤是否为癌变。

检测信用卡交易中的诈骗行为

这里的输入是信用卡交易记录，输出是该交易记录是否可能为诈骗。假设你是信用卡的发行单位，收集数据集意味着需要保存所有的交易，并记录用户是否上报过任何诈骗交易。

在这些例子中需要注意一个有趣的现象，就是虽然输入和输出看起来相当简单，但三个例子中的数据收集过程却大不相同。阅读信封虽然很辛苦，却非常简单，也不用花多少钱。与之相反，获取医学影像和诊断不仅需要昂贵的设备，还需要稀有又昂贵的专家知识，更不要说伦理问题和隐私问题了。在检测信用卡诈骗的例子中，收集数据要容易得多。你的顾客会上报诈骗行为，从而为你提供预期输出。要获取所有欺诈行为和非欺诈行为的输入 / 输出对，你只需等待即可。

本书会讲到的另一类算法是**无监督学习算法**（unsupervised learning algorithm）。在无监督学习中，只有输入数据是已知的，没有为算法提供输出数据。虽然这种算法有许多成功的应用，但理解和评估这些算法往往更加困难。

无监督学习的示例如下。

确定一系列博客文章的主题

如果你有许多文本数据，可能想对其进行汇总，并找到其中共同的主题。事先你可能并不知道都有哪些主题，或者可能有多少个主题。所以输出是未知的。

将客户分成具有相似偏好的群组

给定一组客户记录，你可能想要找出哪些客户比较相似，并判断能否根据相似偏好对这些客户进行分组。对于一家购物网站来说，客户分组可能是“父母”“书虫”或“游戏玩家”。由于你事先并不知道可能有哪些分组，甚至不知道有多少组，所以并不知道输出是什么。

检测网站的异常访问模式

想要识别网站的滥用或 bug，找到异常的访问模式往往是很有用的。每种异常访问模式都互不相同，而且你可能没有任何记录在案的异常行为示例。在这个例子中你只是观察流量，并不知道什么是正常访问行为和异常访问行为，所以这是一个无监督学习问题。

无论是监督学习任务还是无监督学习任务，将输入数据表征为计算机可以理解的形式都是十分重要的。通常来说，将数据想象成表格是很有用的。你想要处理的每一个数据点（每一封电子邮件、每一名客户、每一次交易）对应表格中的一行，描述该数据点的每一项属性（比如客户年龄、交易金额或交易地点）对应表格中的一列。你可能会从年龄、性别、账号创建时间、在你的购物网站上的购买频率等方面来描述用户。你可能会用每一个像素的灰度值来描述肿瘤图像，也可能利用肿瘤的大小、形状和颜色进行描述。

在机器学习中，这里的每个实体或每一行被称为一个**样本**（sample）或数据点，而每一列（用来描述这些实体的属性）则被称为**特征**（feature）。

本书后面会更详细地介绍如何构建良好的数据表征，这被称为**特征提取**（feature extraction）或**特征工程**（feature engineering）。但你应该记住，如果没有数据信息的话，所有机器学习算法都无法做出预测。举个例子，如果你只有病人的姓氏这一个特征，那么任何算法都无法预测其性别。这一信息并未包含在数据中。如果你添加另一个特征，里面包含病人的名字，那么你预测正确的可能性就会变大，因为通过一个人的名字往往可以判断其性别。

1.1.2 熟悉任务和数据

在机器学习过程中，最重要的部分很可能是理解你正在处理的数据，以及这些数据与你想要解决的任务之间的关系。随机选择一个算法并将你的数据输入进去，这种做法是不会有什么用的。在开始构建模型之前，你需要理解数据集的内容。每一种算法的输入数据类型和最适合解决的问题都是不一样的。在构建机器学习解决方案的过程中，你应该给出下列问题的答案，或者至少要将这些问题记在脑中。

- 我想要回答的问题是什么？已经收集到的数据能够回答这个问题吗？
- 要将我的问题表示成机器学习问题，用哪种方法最好？
- 我收集的数据是否足够表达我想要解决的问题？
- 我提取了数据的哪些特征？这些特征能否实现正确的预测？
- 如何衡量应用是否成功？
- 机器学习解决方案与我的研究或商业产品中的其他部分是如何相互影响的？

从更大的层面来看，机器学习算法和方法只是解决特定问题的过程中的一部分，一定要始终牢记整个项目的大局。许多人浪费大量时间构建复杂的机器学习解决方案，最终却发现没有解决正确的问题。

当深入研究机器学习的技术细节时（本书会讲到这些细节），很容易忽视最终目标。我们虽然不会详细讨论上面列出的问题，但仍然鼓励你记住自己在开始构建机器学习模型时做出的假设，无论是明确的还是隐含的假设。

1.2 为何选择Python

Python 已经成为许多数据科学应用的通用语言。它既有通用编程语言的强大功能，也有特定领域脚本语言（比如 MATLAB 或 R）的易用性。Python 有用于数据加载、可视化、统计、自然语言处理、图像处理等各种功能的库。这个大型工具箱为数据科学家提供了大量的通用功能和专用功能。使用 Python 的主要优点之一，就是利用终端或其他类似 Jupyter Notebook 的工具能够直接与代码进行交互；我们很快会讲到 Jupyter Notebook。机器学习和数据分析本质上都是迭代过程，由数据驱动分析。这些过程必须要有快速迭代和易于交互的工具。

作为通用编程语言，Python 还可以用来创建复杂的图形用户界面（graphical user interface，GUI）和 Web 服务，也可以集成到现有系统中。

1.3 scikit-learn

scikit-learn 是一个开源项目，可以免费使用和分发，任何人都可以轻松获取其源代码来查看其背后的原理。scikit-learn 项目正在不断地开发和改进中，它的用户社区非常活跃。它包含许多目前最先进的机器学习算法，每个算法都有详细的文档（http://scikit-learn.org/stable/documentation）。scikit-learn 是一个非常流行的工具，也是最有名的 Python 机器学习库。它广泛应用于工业界和学术界，网上有大量的教程和代码片段。scikit-learn

也可以与其他大量 Python 科学计算工具一起使用，本章后面会讲到相关内容。

在阅读本书的过程中，我们建议你同时浏览 scikit-learn 用户指南（http://scikit-learn.org/stable/user_guide.html）和 API 文档，里面给出了每个算法的更多细节和更多选项。在线文档非常全面，而本书会介绍机器学习的所有必备知识，以便于你深入了解。

安装scikit-learn

scikit-learn 依赖于另外两个 Python 包：NumPy 和 SciPy。若想绘图和进行交互式开发，还应该安装 matplotlib、IPython 和 Jupyter Notebook。我们推荐使用下面三个预先打包的 Python 发行版之一，里面已经装有必要的包。

Anaconda（https://store.continuum.io/cshop/anaconda/）
: 用于大规模数据处理、预测分析和科学计算的 Python 发行版。Anaconda 已经预先安装好 NumPy、SciPy、matplotlib、pandas、IPython、Jupyter Notebook 和 scikit-learn。它可以在 Mac OS、Windows 和 Linux 上运行，是一种非常方便的解决方案。对于尚未安装 Python 科学计算包的人，我们建议使用 Anaconda。Anaconda 现在还免费提供商用的 Intel MKL 库。MKL（在安装 Anaconda 时自动安装）可以使 scikit-learn 中许多算法的速度大大提升。

Enthought Canopy（https://www.enthought.com/products/canopy/）
: 用于科学计算的另一款 Python 发行版。它已经预先装有 NumPy、SciPy、matplotlib、pandas 和 IPython，但免费版没有预先安装 scikit-learn。如果你是能够授予学位的学术机构的成员，可以申请学术许可，免费使用 Enthought Canopy 的付费订阅版。Enthought Canopy 适用于 Python 2.7.x，可以在 Mac OS、Windows 和 Linux 上运行。

Python(x,y)（http://python-xy.github.io/）
: 专门为 Windows 打造的 Python 科学计算免费发行版。Python(x,y) 已经预先装有 NumPy、SciPy、matplotlib、pandas、IPython 和 scikit-learn。

如果你已经安装了 Python，可以用 pip 安装上述所有包：

```
$ pip install numpy scipy matplotlib ipython scikit-learn pandas
```

1.4　必要的库和工具

了解 scikit-learn 及其用法是很重要的，但还有其他一些库也可以改善你的编程体验。scikit-learn 是基于 NumPy 和 SciPy 科学计算库的。除了 NumPy 和 SciPy，我们还会用到 pandas 和 matplotlib。我们还会介绍 Jupyter Notebook，一个基于浏览器的交互编程环境。简单来说，对于这些工具，你应该了解以下内容，以便充分利用 scikit-learn。[1]

注 1：你如果不熟悉 NumPy 或 matplotlib，我们推荐阅读 SciPy 讲稿（http://www.scipy-lectures.org/）的第 1 章。

1.4.1 Jupyter Notebook

Jupyter Notebook 是可以在浏览器中运行代码的交互环境。这个工具在探索性数据分析方面非常有用，在数据科学家中广为使用。虽然 Jupyter Notebook 支持多种编程语言，但我们只需要支持 Python 即可。用 Jupyter Notebook 整合代码、文本和图像非常方便，实际上本书所有内容都是以 Jupyter Notebook 的形式进行编写的。所有代码示例都可以在 GitHub 下载（https://github.com/amueller/introduction_to_ml_with_python）。

1.4.2 NumPy

NumPy 是 Python 科学计算的基础包之一。它的功能包括多维数组、高级数学函数（比如线性代数运算和傅里叶变换），以及伪随机数生成器。

在 `scikit-learn` 中，NumPy 数组是基本数据结构。`scikit-learn` 接受 NumPy 数组格式的数据。你用到的所有数据都必须转换成 NumPy 数组。NumPy 的核心功能是 `ndarray` 类，即多维（*n* 维）数组。数组的所有元素必须是同一类型。NumPy 数组如下所示：

In[2]:

```
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]])
print("x:\n{}".format(x))
```

Out[2]:

```
x:
[[1 2 3]
 [4 5 6]]
```

本书会经常用到 NumPy。对于 NumPy `ndarray` 类的对象，我们将其简称为“NumPy 数组”或“数组”。

1.4.3 SciPy

SciPy 是 Python 中用于科学计算的函数集合。它具有线性代数高级程序、数学函数优化、信号处理、特殊数学函数和统计分布等多项功能。`scikit-learn` 利用 SciPy 中的函数集合来实现算法。对我们来说，SciPy 中最重要的是 `scipy.sparse`：它可以给出**稀疏矩阵**（sparse matrice），稀疏矩阵是 `scikit-learn` 中数据的另一种表示方法。如果想保存一个大部分元素都是 0 的二维数组，就可以使用稀疏矩阵：

In[3]:

```
from scipy import sparse

# 创建一个二维NumPy数组，对角线为1，其余都为0
eye = np.eye(4)
print("NumPy array:\n{}".format(eye))
```

Out[3]:

```
NumPy array:
[[ 1.  0.  0.  0.]
```

```
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

In[4]:

```
# 将NumPy数组转换为CSR格式的SciPy稀疏矩阵
# 只保存非零元素
sparse_matrix = sparse.csr_matrix(eye)
print("\nSciPy sparse CSR matrix:\n{}".format(sparse_matrix))
```

Out[4]:

```
SciPy sparse CSR matrix:
  (0, 0)    1.0
  (1, 1)    1.0
  (2, 2)    1.0
  (3, 3)    1.0
```

通常来说，创建稀疏数据的稠密表示（dense representation）是不可能的（因为太浪费内存），所以我们需要直接创建其稀疏表示（sparse representation）。下面给出的是创建同一稀疏矩阵的方法，用的是 COO 格式：

In[5]:

```
data = np.ones(4)
row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
print("COO representation:\n{}".format(eye_coo))
```

Out[5]:

```
COO representation:
  (0, 0)    1.0
  (1, 1)    1.0
  (2, 2)    1.0
  (3, 3)    1.0
```

关于 SciPy 稀疏矩阵的更多内容可查阅 SciPy 讲稿（http://www.scipy-lectures.org/）。

1.4.4 matplotlib

`matplotlib` 是 Python 主要的科学绘图库，其功能为生成可发布的可视化内容，如折线图、直方图、散点图等。将数据及各种分析可视化，可以让你产生深刻的理解，而我们将用 `matplotlib` 完成所有的可视化内容。在 Jupyter Notebook 中，你可以使用 `%matplotlib notebook` 和 `%matplotlib inline` 命令，将图像直接显示在浏览器中。我们推荐使用 `%matplotlib notebook` 命令，它可以提供交互环境（虽然在写作本书时我们用的是 `%matplotlib inline`）。举个例子，下列代码会生成图 1-1 中的图像：

In[6]:

```
%matplotlib inline
import matplotlib.pyplot as plt

# 在-10和10之间生成一个数列，共100个数
```

```
x = np.linspace(-10, 10, 100)
# 用正弦函数创建第二个数组
y = np.sin(x)
# plot函数绘制一个数组关于另一个数组的折线图
plt.plot(x, y, marker="x")
```

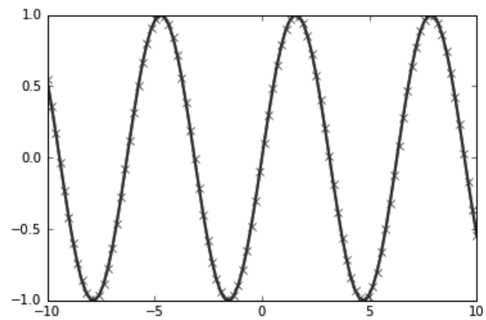


图 1-1：用 matplotlib 画出正弦函数的简单折线图

1.4.5　pandas

pandas 是用于处理和分析数据的 Python 库。它基于一种叫作 DataFrame 的数据结构，这种数据结构模仿了 R 语言中的 DataFrame。简单来说，一个 pandas DataFrame 是一张表格，类似于 Excel 表格。pandas 中包含大量用于修改表格和操作表格的方法，尤其是可以像 SQL 一样对表格进行查询和连接。NumPy 要求数组中的所有元素类型必须完全相同，而 pandas 不是这样，每一列数据的类型可以互不相同（比如整型、日期、浮点数和字符串）。pandas 的另一个强大之处在于，它可以从许多文件格式和数据库中提取数据，如 SQL、Excel 文件和逗号分隔值（CSV）文件。pandas 的详细功能介绍已经超出了本书的范围。但 Wes McKinney 的《Python 数据处理》[2] 一书是很好的参考指南。下面是利用字典创建 DataFrame 的一个小例子：

In[7]:

```
import pandas as pd
from IPython.display import display

# 创建关于人的简单数据集
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location' : ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33]
       }

data_pandas = pd.DataFrame(data)
# IPython.display可以在Jupyter Notebook中打印出“美观的”DataFrame
display(data_pandas)
```

注 2：该书已由人民邮电出版社出版，详见 http://www.ituring.com.cn/book/1819。——编者注

上述代码的输出如下：

	Age	Location	Name
0	24	New York	John
1	13	Paris	Anna
2	53	Berlin	Peter
3	33	London	Linda

查询这个表格的方法有很多种。举个例子：

In[8]:

```
# 选择年龄大于30的所有行
display(data_pandas[data_pandas.Age > 30])
```

输出结果如下：

	Age	Location	Name
2	53	Berlin	Peter
3	33	London	Linda

1.4.6 mglearn

本书的附加代码可以在 GitHub 下载（https://github.com/amueller/introduction_to_ml_with_python）。附加代码不仅包括本书中的所有示例，还包括 `mglearn` 库。 这是我们为本书编写的实用函数库，以免将代码清单与绘图和数据加载的细节混在一起。感兴趣的话，你可以查看仓库中的所有函数，但 `mglearn` 模块的细节并不是本书的重点。如果你在代码中看到了对 `mglearn` 的调用，通常是用来快速美化绘图，或者用于获取一些有趣的数据。

> 本书会频繁使用 NumPy、`matplotlib` 和 `pandas`。所有代码都默认导入了这些库：
>
> ```
> import numpy as np
> import matplotlib.pyplot as plt
> import pandas as pd
> import mglearn
> ```
>
> 我们还假设你在 Jupyter Notebook 中运行代码，并使用 `%matplotlib notebook` 或 `%matplotlib inline` 魔法命令来显示图像。如果你没有使用 Jupyter Notebook 或这些魔法命令，那么就需要调用 `plt.show` 来显示图像。

1.5 Python 2与Python 3的对比

目前 Python 主要有两大版本广为使用：Python 2（更确切地说是 Python 2.7）和 Python 3（写作本书时的最新版本是 Python 3.5）。有时这会造成一些混乱。Python 2 已经停止开发，但由于 Python 3 包含许多重大变化，所以 Python 2 的代码通常无法在 Python 3 中运行。如果你是 Python 新手，或者要从头开发一个新项目，我们强烈推荐使用最新版本的 Python 3，你无需做任何更改。如果你要依赖一个用 Python 2 编写的大型代码库，可以暂时不升级。

但你应该尽快迁移到 Python 3。在编写任何新代码时，想要编写能够在 Python 2 和 Python 3 中同时运行的代码，大多数情况下都是很容易的。[3] 如果你无需与旧版软件进行交互的话，一定要使用 Python 3。本书所有代码在两个版本下都可以运行，但具体的输出在 Python 2 中可能会略有不同。

1.6　本书用到的版本

对于前面提到的这些库，本书用到的版本如下：

In[9]:

```
import sys
print("Python version: {}".format(sys.version))

import pandas as pd
print("pandas version: {}".format(pd.__version__))

import matplotlib
print("matplotlib version: {}".format(matplotlib.__version__))

import numpy as np
print("NumPy version: {}".format(np.__version__))

import scipy as sp
print("SciPy version: {}".format(sp.__version__))

import IPython
print("IPython version: {}".format(IPython.__version__))

import sklearn
print("scikit-learn version: {}".format(sklearn.__version__))
```

Out[9]:

```
Python version: 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
pandas version: 0.18.1
matplotlib version: 1.5.1
NumPy version: 1.11.1
SciPy version: 0.17.1
IPython version: 5.1.0
scikit-learn version: 0.18
```

这些版本不一定要精确匹配，但 `scikit-learn` 的版本不应低于本书使用的版本。

现在都已经安装完毕，我们来学习第一个机器学习应用。

> 本书假设你的 `scikit-learn` 版本不低于 0.18。0.18 版新增了 `model_selection` 模块，如果你用的是较早版本的 `scikit-learn`，那么需要修改从这个模块导入的内容。

注 3：`six` 包（https://pypi.python.org/pypi/six）可以方便地做到这一点。

1.7　第一个应用：鸢尾花分类

本节我们将完成一个简单的机器学习应用，并构建我们的第一个模型。同时还将介绍一些核心概念和术语。

假设有一名植物学爱好者对她发现的鸢尾花的品种很感兴趣。她收集了每朵鸢尾花的一些测量数据：花瓣的长度和宽度以及花萼的长度和宽度，所有测量结果的单位都是厘米（见图 1-2）。

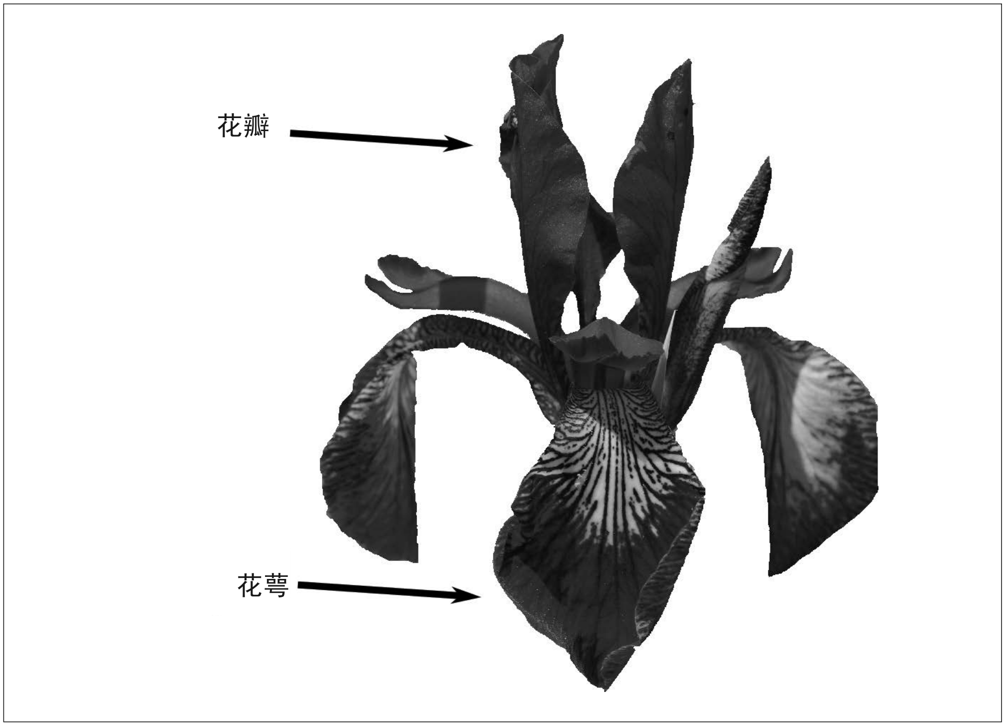


图 1-2：鸢尾花局部

她还有一些鸢尾花的测量数据，这些花之前已经被植物学专家鉴定为属于 setosa、versicolor 或 virginica 三个品种之一。对于这些测量数据，她可以确定每朵鸢尾花所属的品种。我们假设这位植物学爱好者在野外只会遇到这三种鸢尾花。

我们的目标是构建一个机器学习模型，可以从这些已知品种的鸢尾花测量数据中进行学习，从而能够预测新鸢尾花的品种。

因为我们有已知品种的鸢尾花的测量数据，所以这是一个监督学习问题。在这个问题中，我们要在多个选项中预测其中一个（鸢尾花的品种）。这是一个**分类**（classification）问题的示例。可能的输出（鸢尾花的不同品种）叫作**类别**（class）。数据集中的每朵鸢尾花都属于三个类别之一，所以这是一个三分类问题。

单个数据点（一朵鸢尾花）的预期输出是这朵花的品种。对于一个数据点来说，它的品种叫作**标签**（label）。

1.7.1　初识数据

本例中我们用到了鸢尾花（Iris）数据集，这是机器学习和统计学中一个经典的数据集。它包含在 `scikit-learn` 的 `datasets` 模块中。我们可以调用 `load_iris` 函数来加载数据：

In[10]:

```
from sklearn.datasets import load_iris
iris_dataset = load_iris()
```

`load_iris` 返回的 `iris` 对象是一个 `Bunch` 对象，与字典非常相似，里面包含键和值：

In[11]:

```
print("Keys of iris_dataset: \n{}".format(iris_dataset.keys()))
```

Out[11]:

```
Keys of iris_dataset:
dict_keys(['target_names', 'feature_names', 'DESCR', 'data', 'target'])
```

`DESCR` 键对应的值是数据集的简要说明。我们这里给出说明的开头部分（你可以自己查看其余的内容）：

In[12]:

```
print(iris_dataset['DESCR'][:193] + "\n...")
```

Out[12]:

```
Iris Plants Database
====================

Notes
----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive att
...
----
```

`target_names` 键对应的值是一个字符串数组，里面包含我们要预测的花的品种：

In[13]:

```
print("Target names: {}".format(iris_dataset['target_names']))
```

Out[13]:

```
Target names: ['setosa' 'versicolor' 'virginica']
```

`feature_names` 键对应的值是一个字符串列表，对每一个特征进行了说明：

In[14]:

```
print("Feature names: \n{}".format(iris_dataset['feature_names']))
```

Out[14]:

```
Feature names:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
 'petal width (cm)']
```

数据包含在 `target` 和 `data` 字段中。`data` 里面是花萼长度、花萼宽度、花瓣长度、花瓣宽度的测量数据，格式为 NumPy 数组：

In[15]:

```
print("Type of data: {}".format(type(iris_dataset['data'])))
```

Out[15]:

```
Type of data: <class 'numpy.ndarray'>
```

`data` 数组的每一行对应一朵花，列代表每朵花的四个测量数据：

In[16]:

```
print("Shape of data: {}".format(iris_dataset['data'].shape))
```

Out[16]:

```
Shape of data: (150, 4)
```

可以看出，数组中包含 150 朵不同的花的测量数据。前面说过，机器学习中的个体叫作**样本**（sample），其属性叫作**特征**（feature）。`data` 数组的**形状**（shape）是样本数乘以特征数。这是 `scikit-learn` 中的约定，你的数据形状应始终遵循这个约定。下面给出前 5 个样本的特征数值：

In[17]:

```
print("First five rows of data:\n{}".format(iris_dataset['data'][:5]))
```

Out[17]:

```
First five rows of data:
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]]
```

从数据中可以看出，前 5 朵花的花瓣宽度都是 0.2cm，第一朵花的花萼最长，是 5.1cm。

`target` 数组包含的是测量过的每朵花的品种，也是一个 NumPy 数组：

In[18]:

```
print("Type of target: {}".format(type(iris_dataset['target'])))
```

Out[18]:

```
Type of target: <class 'numpy.ndarray'>
```

`target` 是一维数组，每朵花对应其中一个数据：

In[19]:

```
print("Shape of target: {}".format(iris_dataset['target'].shape))
```

Out[19]:

```
Shape of target: (150,)
```

品种被转换成从 0 到 2 的整数：

In[20]:

```
print("Target:\n{}".format(iris_dataset['target']))
```

Out[20]:

```
Target:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

上述数字的代表含义由 `iris['target_names']` 数组给出：0 代表 setosa，1 代表 versicolor，2 代表 virginica。

1.7.2　衡量模型是否成功：训练数据与测试数据

我们想要利用这些数据构建一个机器学习模型，用于预测新测量的鸢尾花的品种。但在将模型应用于新的测量数据之前，我们需要知道模型是否有效，也就是说，我们是否应该相信它的预测结果。

不幸的是，我们不能将用于构建模型的数据用于评估模型。因为我们的模型会一直记住整个训练集，所以对于训练集中的任何数据点总会预测正确的标签。这种“记忆”无法告诉我们模型的**泛化**（generalize）能力如何（换句话说，在新数据上能否正确预测）。

我们要用新数据来评估模型的性能。新数据是指模型之前没有见过的数据，而我们有这些新数据的标签。通常的做法是将收集好的带标签数据（此例中是 150 朵花的测量数据）分成两部分。一部分数据用于构建机器学习模型，叫作**训练数据**（training data）或**训练集**（training set）。其余的数据用来评估模型性能，叫作**测试数据**（test data）、**测试集**（test set）或**留出集**（hold-out set）。

`scikit-learn` 中的 `train_test_split` 函数可以打乱数据集并进行拆分。这个函数将 75% 的行数据及对应标签作为训练集，剩下 25% 的数据及其标签作为测试集。训练集与测试集的分配比例可以是随意的，但使用 25% 的数据作为测试集是很好的经验法则。

`scikit-learn` 中的数据通常用大写的 `X` 表示，而标签用小写的 `y` 表示。这是受到了数学标准公式 $f(x)=y$ 的启发，其中 x 是函数的输入，y 是输出。我们用大写的 `X` 是因为数据是一个二维数组（矩阵），用小写的 `y` 是因为目标是一个一维数组（向量），这也是数学中的约定。

对数据调用 `train_test_split`，并对输出结果采用下面这种命名方法：

In[21]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)
```

在对数据进行拆分之前，train_test_split 函数利用伪随机数生成器将数据集打乱。如果我们只是将最后 25% 的数据作为测试集，那么所有数据点的标签都是 2，因为数据点是按标签排序的（参见之前 iris['target'] 的输出）。测试集中只有三个类别之一，这无法告诉我们模型的泛化能力如何，所以我们将数据打乱，确保测试集中包含所有类别的数据。

为了确保多次运行同一函数能够得到相同的输出，我们利用 random_state 参数指定了随机数生成器的种子。这样函数输出就是固定不变的，所以这行代码的输出始终相同。本书用到随机过程时，都会用这种方法指定 random_state。

train_test_split 函数的输出为 X_train、X_test、y_train 和 y_test，它们都是 NumPy 数组。X_train 包含 75% 的行数据，X_test 包含剩下的 25%：

In[22]:

```
print("X_train shape: {}".format(X_train.shape))
print("y_train shape: {}".format(y_train.shape))
```

Out[22]:

```
X_train shape: (112, 4)
y_train shape: (112,)
```

In[23]:

```
print("X_test shape: {}".format(X_test.shape))
print("y_test shape: {}".format(y_test.shape))
```

Out[23]:

```
X_test shape: (38, 4)
y_test shape: (38,)
```

1.7.3　要事第一：观察数据

在构建机器学习模型之前，通常最好检查一下数据，看看如果不用机器学习能不能轻松完成任务，或者需要的信息有没有包含在数据中。

此外，检查数据也是发现异常值和特殊值的好方法。举个例子，可能有些鸢尾花的测量单位是英寸而不是厘米。在现实世界中，经常会遇到不一致的数据和意料之外的测量数据。

检查数据的最佳方法之一就是将其可视化。一种可视化方法是绘制**散点图**（scatter plot）。数据散点图将一个特征作为 x 轴，另一个特征作为 y 轴，将每一个数据点绘制为图上的一个点。不幸的是，计算机屏幕只有两个维度，所以我们一次只能绘制两个特征（也可能是 3 个）。用这种方法难以对多于 3 个特征的数据集作图。解决这个问题的一种方法是绘制**散点图矩阵**（pair plot），从而可以两两查看所有的特征。如果特征数不多的话，比如我们这里有 4 个，这种方法是很合理的。但是你应该记住，散点图矩阵无法同时显示所有特征之间的关系，所以这种可视化方法可能无法展示数据的某些有趣内容。

图 1-3 是训练集中特征的散点图矩阵。数据点的颜色与鸢尾花的品种相对应。为了绘制这张图，我们首先将 NumPy 数组转换成 pandas DataFrame。pandas 有一个绘制散点图矩阵的函数，叫作 scatter_matrix。矩阵的对角线是每个特征的直方图：

In[24]:

```
# 利用X_train中的数据创建DataFrame
# 利用iris_dataset.feature_names中的字符串对数据列进行标记
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
# 利用DataFrame创建散点图矩阵，按y_train着色
grr = pd.scatter_matrix(iris_dataframe, c=y_train, figsize=(15, 15), marker='o',
                        hist_kwds={'bins': 20}, s=60, alpha=.8, cmap=mglearn.cm3)
```

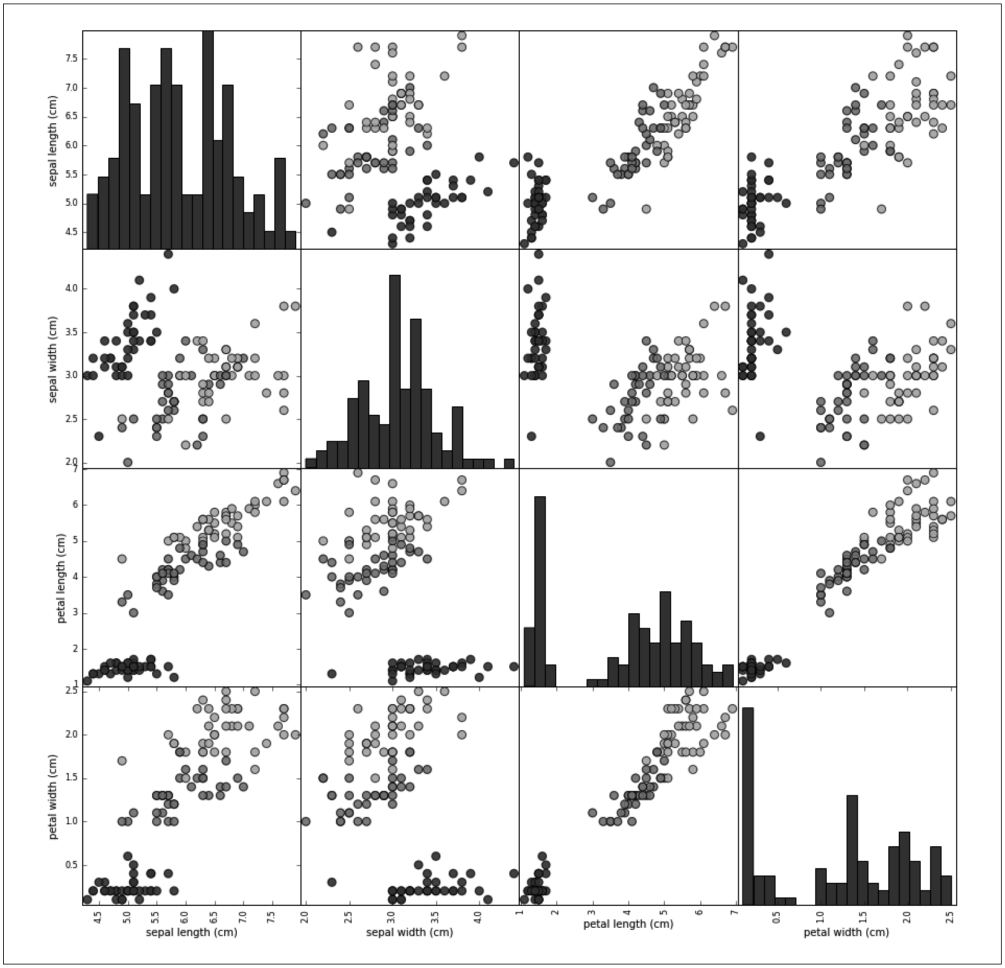


图 1-3：Iris 数据集的散点图矩阵，按类别标签着色

从图中可以看出，利用花瓣和花萼的测量数据基本可以将三个类别区分开。这说明机器学习模型很可能可以学会区分它们。

1.7.4　构建第一个模型：k近邻算法

现在我们可以开始构建真实的机器学习模型了。scikit-learn 中有许多可用的分类算法。

这里我们用的是 k 近邻分类器，这是一个很容易理解的算法。构建此模型只需要保存训练集即可。要对一个新的数据点做出预测，算法会在训练集中寻找与这个新数据点距离最近的数据点，然后将找到的数据点的标签赋值给这个新数据点。

k 近邻算法中 k 的含义是，我们可以考虑训练集中与新数据点最近的任意 *k* 个邻居（比如说，距离最近的 3 个或 5 个邻居），而不是只考虑最近的那一个。然后，我们可以用这些邻居中数量最多的类别做出预测。第 2 章会进一步介绍这个算法的细节，现在我们只考虑一个邻居的情况。

`scikit-learn` 中所有的机器学习模型都在各自的类中实现，这些类被称为 `Estimator` 类。k 近邻分类算法是在 `neighbors` 模块的 `KNeighborsClassifier` 类中实现的。我们需要将这个类实例化为一个对象，然后才能使用这个模型。这时我们需要设置模型的参数。`KNeighborsClassifier` 最重要的参数就是邻居的数目，这里我们设为 `1`：

In[25]:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

`knn` 对象对算法进行了封装，既包括用训练数据构建模型的算法，也包括对新数据点进行预测的算法。它还包括算法从训练数据中提取的信息。对于 `KNeighborsClassifier` 来说，里面只保存了训练集。

想要基于训练集来构建模型，需要调用 `knn` 对象的 `fit` 方法，输入参数为 `X_train` 和 `y_train`，二者都是 NumPy 数组，前者包含训练数据，后者包含相应的训练标签：

In[26]:

```
knn.fit(X_train, y_train)
```

Out[26]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

`fit` 方法返回的是 `knn` 对象本身并做原处修改，因此我们得到了分类器的字符串表示。从中可以看出构建模型时用到的参数。几乎所有参数都是默认值，但你也会注意到 `n_neighbors=1`，这是我们传入的参数。`scikit-learn` 中的大多数模型都有很多参数，但多用于速度优化或非常特殊的用途。你无需关注这个字符串表示中的其他参数。打印 `scikit-learn` 模型会生成非常长的字符串，但不要被它吓到。我们会在第 2 章讲到所有重要的参数。在本书的其他章节中，我们不会给出 `fit` 的输出，因为里面没有包含任何新的信息。

1.7.5 做出预测

现在我们可以用这个模型对新数据进行预测了，我们可能并不知道这些新数据的正确标签。想象一下，我们在野外发现了一朵鸢尾花，花萼长 5cm 宽 2.9cm，花瓣长 1cm 宽 0.2cm。这朵鸢尾花属于哪个品种？我们可以将这些数据放在一个 NumPy 数组中，再次计算形状，数组形状为样本数（1）乘以特征数（4）：

In[27]:

```
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape: {}".format(X_new.shape))
```

Out[27]:

```
X_new.shape: (1, 4)
```

注意，我们将这朵花的测量数据转换为二维 NumPy 数组的一行，这是因为 scikit-learn 的输入数据必须是二维数组。

我们调用 knn 对象的 predict 方法来进行预测：

In[28]:

```
prediction = knn.predict(X_new)
print("Prediction: {}".format(prediction))
print("Predicted target name: {}".format(
       iris_dataset['target_names'][prediction]))
```

Out[28]:

```
Prediction: [0]
Predicted target name: ['setosa']
```

根据我们模型的预测，这朵新的鸢尾花属于类别 0，也就是说它属于 setosa 品种。但我们怎么知道能否相信这个模型呢？我们并不知道这个样本的实际品种，这也是我们构建模型的重点啊！

1.7.6 评估模型

这里需要用到之前创建的测试集。这些数据没有用于构建模型，但我们知道测试集中每朵鸢尾花的实际品种。

因此，我们可以对测试数据中的每朵鸢尾花进行预测，并将预测结果与标签（已知的品种）进行对比。我们可以通过计算**精度**（accuracy）来衡量模型的优劣，精度就是品种预测正确的花所占的比例：

In[29]:

```
y_pred = knn.predict(X_test)
print("Test set predictions:\n {}".format(y_pred))
```

Out[29]:

```
Test set predictions:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0 2]
```

In[30]:

```
print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))
```

Out[30]:

```
Test set score: 0.97
```

我们还可以使用 knn 对象的 score 方法来计算测试集的精度：

In[31]:

```
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[31]:

```
Test set score: 0.97
```

对于这个模型来说，测试集的精度约为 0.97，也就是说，对于测试集中的鸢尾花，我们的预测有 97% 是正确的。根据一些数学假设，对于新的鸢尾花，可以认为我们的模型预测结果有 97% 都是正确的。对于我们的植物学爱好者应用程序来说，高精度意味着模型足够可信，可以使用。在后续章节中，我们将讨论提高性能的方法，以及模型调参时的注意事项。

1.8　小结与展望

总结一下本章所学的内容。我们首先简要介绍了机器学习及其应用，然后讨论了监督学习和无监督学习之间的区别，并简要介绍了本书将会用到的工具。随后，我们构思了一项任务，要利用鸢尾花的物理测量数据来预测其品种。我们在构建模型时用到了由专家标注过的测量数据集，专家已经给出了花的正确品种，因此这是一个监督学习问题。一共有三个品种：setosa、versicolor 或 virginica，因此这是一个三分类问题。在分类问题中，可能的品种被称为**类别**（class），每朵花的品种被称为它的**标签**（label）。

鸢尾花（Iris）数据集包含两个 NumPy 数组：一个包含数据，在 `scikit-learn` 中被称为 `X`；一个包含正确的输出或预期输出，被称为 `y`。数组 `X` 是特征的二维数组，每个数据点对应一行，每个特征对应一列。数组 `y` 是一维数组，里面包含一个类别标签，对每个样本都是一个 0 到 2 之间的整数。

我们将数据集分成**训练集**（training set）和**测试集**（test set），前者用于构建模型，后者用于评估模型对前所未见的新数据的泛化能力。

我们选择了 k 近邻分类算法，根据新数据点在训练集中距离最近的邻居来进行预测。该算法在 `KNeighborsClassifier` 类中实现，里面既包含构建模型的算法，也包含利用模型进行预测的算法。我们将类实例化，并设定参数。然后调用 `fit` 方法来构建模型，传入训练数据（`X_train`）和训练输出（`y_trian`）作为参数。我们用 `score` 方法来评估模型，该方法计算的是模型精度。我们将 `score` 方法用于测试集数据和测试集标签，得出模型的精度约为 97%，也就是说，该模型在测试集上 97% 的预测都是正确的。

这让我们有信心将模型应用于新数据（在我们的例子中是新花的测量数据），并相信模型在约 97% 的情况下都是正确的。

下面汇总了整个训练和评估过程所必需的代码：

In[32]:

```
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
```

```
knn.fit(X_train, y_train)

print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[32]:

```
Test set score: 0.97
```

这个代码片段包含了应用 scikit-learn 中任何机器学习算法的核心代码。fit、predict 和 score 方法是 scikit-learn 监督学习模型中最常用的接口。学完本章介绍的概念，你可以将这些模型应用到许多机器学习任务上。下一章，我们会更深入地介绍 scikit-learn 中各种类型的监督学习模型，以及这些模型的正确使用方法。

第 2 章

监督学习

前面说过，监督学习是最常用也是最成功的机器学习类型之一。本章将会详细介绍监督学习，并解释几种常用的监督学习算法。我们在第 1 章已经见过一个监督学习的应用：利用物理测量数据将鸢尾花分成几个品种。

记住，每当想要根据给定输入预测某个结果，并且还有输入 / 输出对的示例时，都应该使用监督学习。这些输入 / 输出对构成了训练集，我们利用它来构建机器学习模型。我们的目标是对从未见过的新数据做出准确预测。监督学习通常需要人力来构建训练集，但之后的任务本来非常费力甚至无法完成，现在却可以自动完成，通常速度也更快。

2.1　分类与回归

监督机器学习问题主要有两种，分别叫作**分类**（classification）与**回归**（regression）。

分类问题的目标是预测**类别标签**（class label），这些标签来自预定义的可选列表。第 1 章讲过一个例子，即将鸢尾花分到三个可能的品种之一。分类问题有时可分为**二分类**（binary classification，在两个类别之间进行区分的一种特殊情况）和**多分类**（multiclass classification，在两个以上的类别之间进行区分）。你可以将二分类看作是尝试回答一道是 / 否问题。将电子邮件分为垃圾邮件和非垃圾邮件就是二分类问题的实例。在这个二分类任务中，要问的是 / 否问题为："这封电子邮件是垃圾邮件吗？"

在二分类问题中，我们通常将其中一个类别称为**正类**（positive class），另一个类别称为**反类**（negative class）。这里的"正"并不代表好的方面或正数，而是代表研究对象。因此在寻找垃圾邮件时，"正"可能指的是垃圾邮件这一类别。将两个类别中的哪一个作为"正类"，往往是主观判断，与具体的领域有关。

另一方面，鸢尾花的例子则属于多分类问题。另一个多分类的例子是根据网站上的文本预

测网站所用的语言。这里的类别就是预定义的语言列表。

回归任务的目标是预测一个连续值，编程术语叫作**浮点数**（floating-point number），数学术语叫作**实数**（real number）。根据教育水平、年龄和居住地来预测一个人的年收入，这就是回归的一个例子。在预测收入时，预测值是一个**金额**（amount），可以在给定范围内任意取值。回归任务的另一个例子是，根据上一年的产量、天气和农场员工数等属性来预测玉米农场的产量。同样，产量也可以取任意数值。

区分分类任务和回归任务有一个简单方法，就是问一个问题：输出是否具有某种连续性。如果在可能的结果之间具有连续性，那么它就是一个回归问题。想想预测年收入的例子。输出具有非常明显的连续性。一年赚 40 000 美元还是 40 001 美元并没有实质差别，即使两者金额不同。如果我们的算法在本应预测 40 000 美元时的预测结果是 39 999 美元或 40 001 美元，不必过分在意。

与此相反，对于识别网站语言的任务（这是一个分类问题）来说，并不存在程度问题。网站使用的要么是这种语言，要么是那种语言。在语言之间不存在连续性，在英语和法语**之间**不存在其他语言。[1]

2.2 泛化、过拟合与欠拟合

在监督学习中，我们想要在训练数据上构建模型，然后能够对没见过的新数据（这些新数据与训练集具有相同的特性）做出准确预测。如果一个模型能够对没见过的数据做出准确预测，我们就说它能够从训练集**泛化**（generalize）到测试集。我们想要构建一个泛化精度尽可能高的模型。

通常来说，我们构建模型，使其在训练集上能够做出准确预测。如果训练集和测试集足够相似，我们预计模型在测试集上也能做出准确预测。不过在某些情况下这一点并不成立。例如，如果我们可以构建非常复杂的模型，那么在训练集上的精度可以想多高就多高。

为了说明这一点，我们来看一个虚构的例子。比如有一个新手数据科学家，已知之前船的买家记录和对买船不感兴趣的顾客记录，想要预测某个顾客是否会买船。[2] 目标是向可能购买的人发送促销电子邮件，而不去打扰那些不感兴趣的顾客。

假设我们有顾客记录，如表 2-1 所示。

表2-1：顾客数据示例

年龄	拥有的小汽车数量	是否有房子	子女数量	婚姻状况	是否养狗	是否买过船
66	1	是	2	丧偶	否	是
52	2	是	3	已婚	否	是
22	0	否	0	已婚	是	否
25	1	否	1	单身	否	否

注 1：请语言学家原谅我们将语言的表示方式简化为独特而又确定的实体。

注 2：在现实世界中，这实际上是一个非常复杂的问题。虽然我们知道其他顾客还没有从我们这里买过船，但他们可能已经在其他人那里买过了，或者仍在存钱并打算将来再买。

（续）

年龄	拥有的小汽车数量	是否有房子	子女数量	婚姻状况	是否养狗	是否买过船
44	0	否	2	离异	是	否
39	1	是	2	已婚	是	否
26	1	否	2	单身	否	否
40	3	是	1	已婚	是	否
53	2	是	2	离异	否	是
64	2	是	3	离异	否	否
58	2	是	2	已婚	是	是
33	1	否	1	单身	否	否

对数据观察一段时间之后，我们的新手数据科学家发现了以下规律："如果顾客年龄大于 45 岁，并且子女少于 3 个或没有离婚，那么他就想要买船。"如果你问他这个规律的效果如何，我们的数据科学家会回答："100% 准确！"的确，对于表中的数据，这条规律完全正确。我们还可以发现好多规律，都可以完美解释这个数据集中的某人是否想要买船。数据中的年龄都没有重复，因此我们可以这样说：66、52、53 和 58 岁的人想要买船，而其他年龄的人都不想买。虽然我们可以编出许多条适用于这个数据集的规律，但要记住，我们感兴趣的并不是对这个数据集进行预测，我们已经知道这些顾客的答案。我们想知道**新顾客**是否可能会买船。因此，我们想要找到一条适用于新顾客的规律，而在训练集上实现 100% 的精度对此并没有帮助。我们可能认为数据科学家发现的规律无法适用于新顾客。它看起来过于复杂，而且只有很少的数据支持。例如，规律里"或没有离婚"这一条对应的只有一名顾客。

判断一个算法在新数据上表现好坏的唯一度量，就是在测试集上的评估。然而从直觉上看[3]，我们认为简单的模型对新数据的泛化能力更好。如果规律是"年龄大于 50 岁的人想要买船"，并且这可以解释所有顾客的行为，那么我们将更相信这条规律，而不是与年龄、子女和婚姻状况都有关系的那条规律。因此，我们总想找到最简单的模型。构建一个对现有信息量来说过于复杂的模型，正如我们的新手数据科学家做的那样，这被称为**过拟合**（overfitting）。如果你在拟合模型时过分关注训练集的细节，得到了一个在训练集上表现很好、但不能泛化到新数据上的模型，那么就存在过拟合。与之相反，如果你的模型过于简单——比如说，"有房子的人都买船"——那么你可能无法抓住数据的全部内容以及数据中的变化，你的模型甚至在训练集上的表现就很差。选择过于简单的模型被称为**欠拟合**（underfitting）。

我们的模型越复杂，在训练数据上的预测结果就越好。但是，如果我们的模型过于复杂，我们开始过多关注训练集中每个单独的数据点，模型就不能很好地泛化到新数据上。

二者之间存在一个最佳位置，可以得到最好的泛化性能。这就是我们想要的模型。

图 2-1 给出了过拟合与欠拟合之间的权衡。

注 3：在数学上也可以证明这一点。

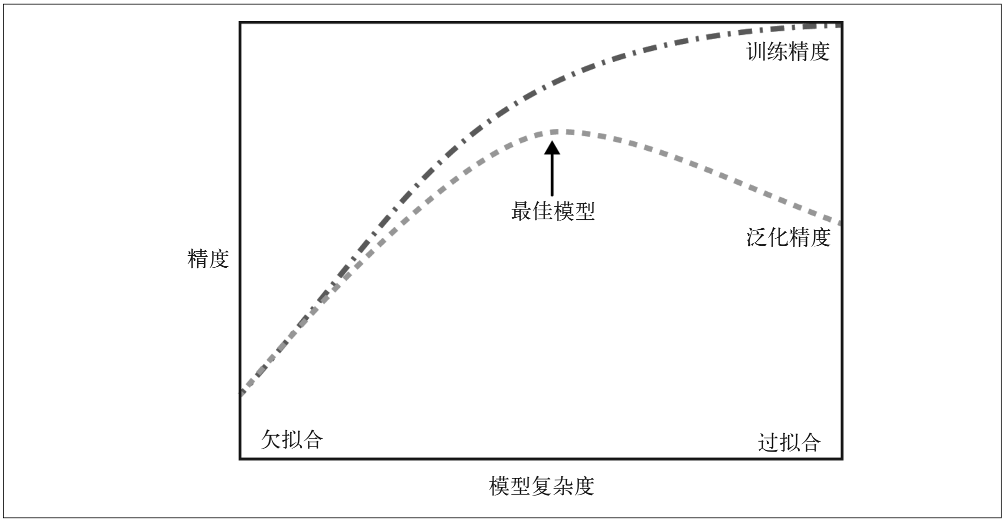


图 2-1：模型复杂度与训练精度和测试精度之间的权衡

模型复杂度与数据集大小的关系

需要注意，模型复杂度与训练数据集中输入的变化密切相关：数据集中包含的数据点的变化范围越大，在不发生过拟合的前提下你可以使用的模型就越复杂。通常来说，收集更多的数据点可以有更大的变化范围，所以更大的数据集可以用来构建更复杂的模型。但是，仅复制相同的数据点或收集非常相似的数据是无济于事的。

回到前面卖船的例子，如果我们查看了 10 000 多行的顾客数据，并且所有数据都符合这条规律："如果顾客年龄大于 45 岁，并且子女少于 3 个或没有离婚，那么他就想要买船"，那么我们就更有可能相信这是一条有效的规律，比从表 2-1 中仅 12 行数据得出来的更为可信。

收集更多数据，适当构建更复杂的模型，对监督学习任务往往特别有用。本书主要关注固定大小的数据集。在现实世界中，你往往能够决定收集多少数据，这可能比模型调参更为有效。永远不要低估更多数据的力量！

2.3　监督学习算法

现在开始介绍最常用的机器学习算法，并解释这些算法如何从数据中学习以及如何预测。我们还会讨论每个模型的复杂度如何变化，并概述每个算法如何构建模型。我们将说明每个算法的优点和缺点，以及它们最适应用于哪类数据。此外还会解释最重要的参数和选项的含义。[4] 许多算法都有分类和回归两种形式，两者我们都会讲到。

注 4：讲解所有的参数和选项超出了本书范围，你可以参阅 `scikit-learn` 文档（http://scikit-learn.org/stable/documentation）来了解更多细节。

没有必要通读每个算法的详细描述，但理解模型可以让你更好地理解机器学习算法的各种工作原理。本章还可以用作参考指南，当你不确定某个算法的工作原理时，就可以回来查看本章内容。

2.3.1 一些样本数据集

我们将使用一些数据集来说明不同的算法。其中一些数据集很小，而且是模拟的，其目的是强调算法的某个特定方面。其他数据集都是现实世界的大型数据集。

一个模拟的二分类数据集示例是 `forge` 数据集，它有两个特征。下列代码将绘制一个散点图（图 2-2），将此数据集的所有数据点可视化。图像以第一个特征为 *x* 轴，第二个特征为 *y* 轴。正如其他散点图那样，每个数据点对应图像中的一点。每个点的颜色和形状对应其类别：

In[2]:

```
# 生成数据集
X, y = mglearn.datasets.make_forge()
# 数据集绘图
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Class 0", "Class 1"], loc=4)
plt.xlabel("First feature")
plt.ylabel("Second feature")
print("X.shape: {}".format(X.shape))
```

Out[2]:

```
X.shape: (26, 2)
```

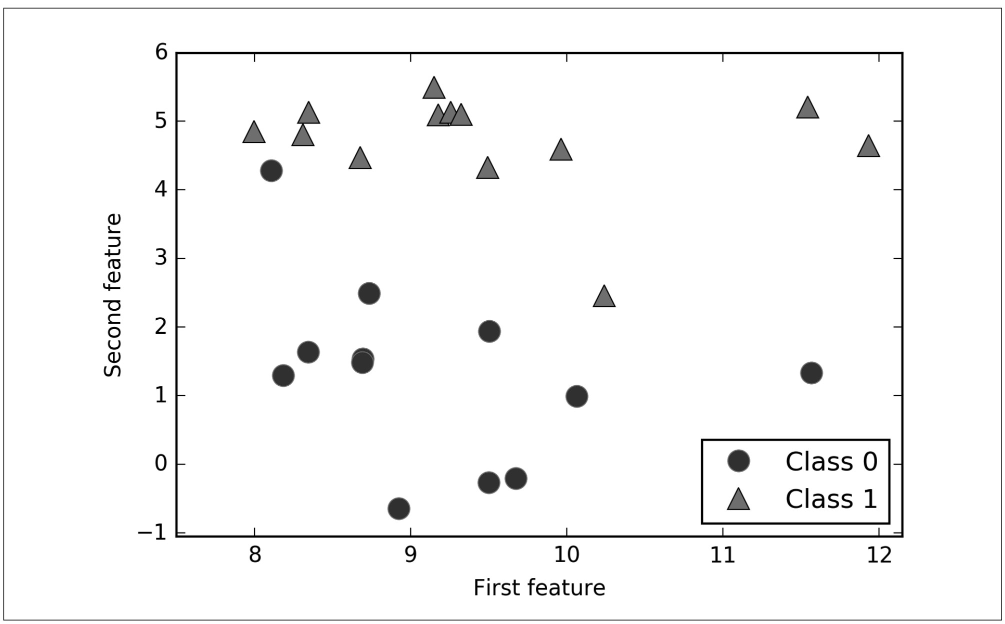


图 2-2：`forge` 数据集的散点图

从 X.shape 可以看出，这个数据集包含 26 个数据点和 2 个特征。

我们用模拟的 wave 数据集来说明回归算法。wave 数据集只有一个输入特征和一个连续的目标变量（或**响应**），后者是模型想要预测的对象。下面绘制的图像（图 2-3）中单一特征位于 *x* 轴，回归目标（输出）位于 *y* 轴：

In[3]:

```
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Feature")
plt.ylabel("Target")
```

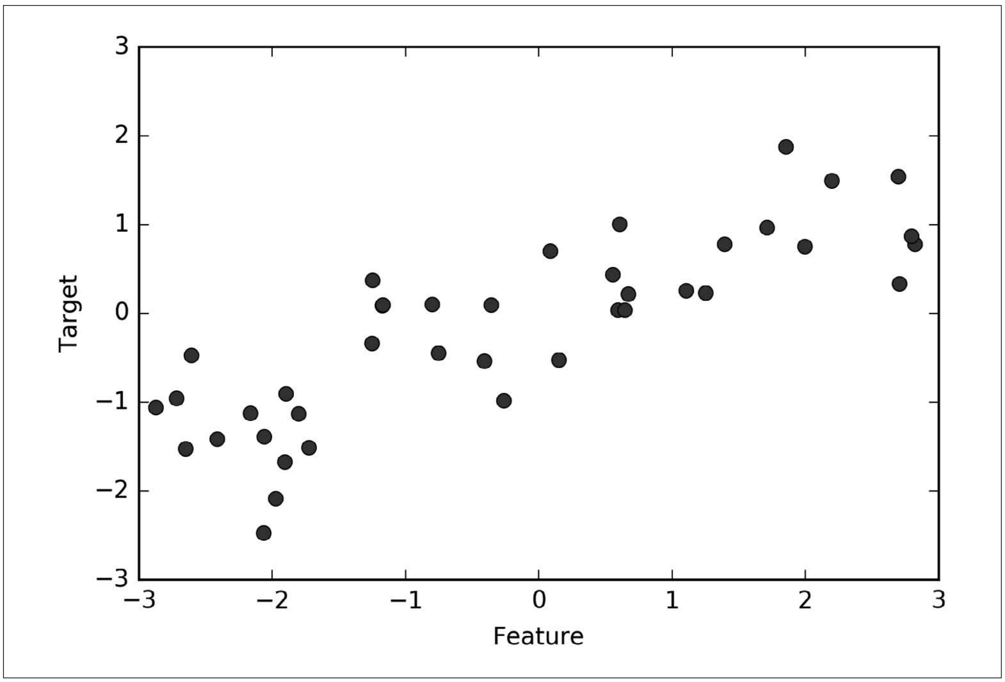


图 2-3：wave 数据集的图像，*x* 轴表示特征，*y* 轴表示回归目标

我们之所以使用这些非常简单的低维数据集，是因为它们的可视化非常简单——书页只有两个维度，所以很难展示特征数超过两个的数据。从特征较少的数据集（也叫**低维**数据集）中得出的结论可能并不适用于特征较多的数据集（也叫**高维**数据集）。只要你记住这一点，那么在低维数据集上研究算法也是很有启发的。

除了上面这些小型的模拟的数据集，我们还将补充两个现实世界中的数据集，它们都包含在 scikit-learn 中。其中一个是威斯康星州乳腺癌数据集（简称 cancer），里面记录了乳腺癌肿瘤的临床测量数据。每个肿瘤都被标记为“良性”（benign，表示无害肿瘤）或“恶性”（malignant，表示癌性肿瘤），其任务是基于人体组织的测量数据来学习预测肿瘤是否为恶性。

可以用 scikit-learn 模块的 load_breast_cancer 函数来加载数据：

In[4]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): \n{}".format(cancer.keys()))
```

Out[4]:

```
cancer.keys():
dict_keys(['feature_names', 'data', 'DESCR', 'target', 'target_names'])
```

> 包含在 scikit-learn 中的数据集通常被保存为 Bunch 对象，里面包含真实数据以及一些数据集信息。关于 Bunch 对象，你只需要知道它与字典很相似，而且还有一个额外的好处，就是你可以用点操作符来访问对象的值（比如用 bunch.key 来代替 bunch['key']）。

这个数据集共包含 569 个数据点，每个数据点有 30 个特征：

In[5]:

```
print("Shape of cancer data: {}".format(cancer.data.shape))
```

Out[5]:

```
Shape of cancer data: (569, 30)
```

在 569 个数据点中，212 个被标记为恶性，357 个被标记为良性：

In[6]:

```
print("Sample counts per class:\n{}".format(
      {n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))
```

Out[6]:

```
Sample counts per class:
{'benign': 357, 'malignant': 212}
```

为了得到每个特征的语义说明，我们可以看一下 feature_names 属性：

In[7]:

```
print("Feature names:\n{}".format(cancer.feature_names))
```

Out[7]:

```
Feature names:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

感兴趣的话，你可以阅读 cancer.DESCR 来了解数据的更多信息。

我们还会用到一个现实世界中的回归数据集，即波士顿房价数据集。与这个数据集相关的任务是，利用犯罪率、是否邻近查尔斯河、公路可达性等信息，来预测 20 世纪 70 年代波士顿地区房屋价格的中位数。这个数据集包含 506 个数据点和 13 个特征：

In[8]:
```
from sklearn.datasets import load_boston
boston = load_boston()
print("Data shape: {}".format(boston.data.shape))
```

Out[8]:
```
Data shape: (506, 13)
```

同样，你可以阅读 `boston` 对象的 `DESCR` 属性来了解数据集的更多信息。对于我们的目的而言，我们需要扩展这个数据集，输入特征不仅包括这 13 个测量结果，还包括这些特征之间的乘积（也叫**交互项**）。换句话说，我们不仅将犯罪率和公路可达性作为特征，还将犯罪率和公路可达性的乘积作为特征。像这样包含导出特征的方法叫作**特征工程**（feature engineering），将在第 4 章中详细讲述。这个导出的数据集可以用 `load_extended_boston` 函数加载：

In[9]:
```
X, y = mglearn.datasets.load_extended_boston()
print("X.shape: {}".format(X.shape))
```

Out[9]:
```
X.shape: (506, 104)
```

最初的 13 个特征加上这 13 个特征两两组合（有放回）得到的 91 个特征，一共有 104 个特征。[5]

我们将利用这些数据集对不同机器学习算法的性质进行解释说明。但目前来说，先来看算法本身。首先重新学习上一章见过的 k 近邻（k-NN）算法。

2.3.2　k近邻

k-NN 算法可以说是最简单的机器学习算法。构建模型只需要保存训练数据集即可。想要对新数据点做出预测，算法会在训练数据集中找到最近的数据点，也就是它的“最近邻”。

1. k近邻分类

k-NN 算法最简单的版本只考虑一个最近邻，也就是与我们想要预测的数据点最近的训练数据点。预测结果就是这个训练数据点的已知输出。图 2-4 给出了这种分类方法在 `forge` 数据集上的应用：

In[10]:
```
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

注 5：第 1 个特征可以与 13 个特征相乘，第 2 个可以与 12 个特征相乘（除了第 1 个），第 3 个可以与 11 个特征相乘……依次相加，$13 + 12 + 11 + \cdots + 1 = 91$。

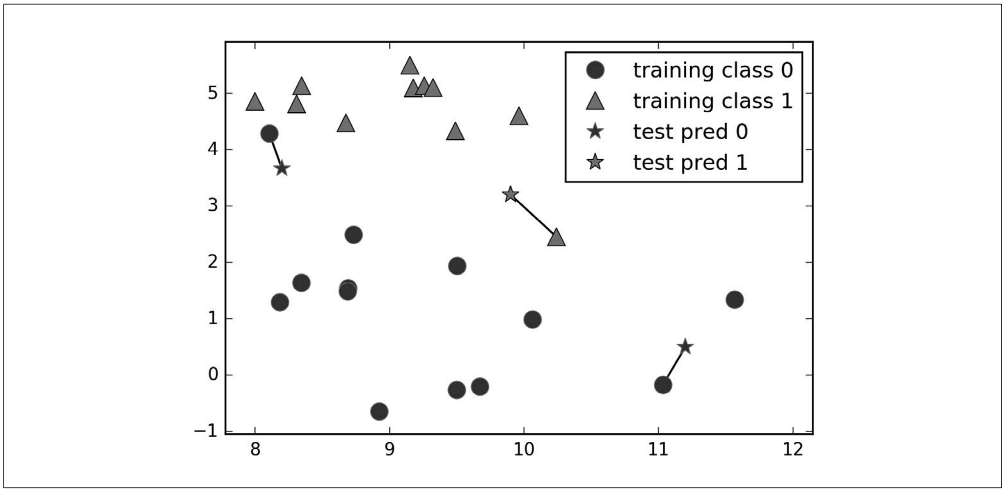


图 2-4：单一最近邻模型对 forge 数据集的预测结果

这里我们添加了 3 个新数据点（用五角星表示）。对于每个新数据点，我们标记了训练集中与它最近的点。单一最近邻算法的预测结果就是那个点的标签（对应五角星的颜色）。

除了仅考虑最近邻，我还可以考虑任意个（*k* 个）邻居。这也是 k 近邻算法名字的来历。在考虑多于一个邻居的情况时，我们用“投票法”（voting）来指定标签。也就是说，对于每个测试点，我们数一数多少个邻居属于类别 0，多少个邻居属于类别 1。然后将出现次数更多的类别（也就是 *k* 个近邻中占多数的类别）作为预测结果。下面的例子（图 2-5）用到了 3 个近邻：

In[11]:

```
mglearn.plots.plot_knn_classification(n_neighbors=3)
```

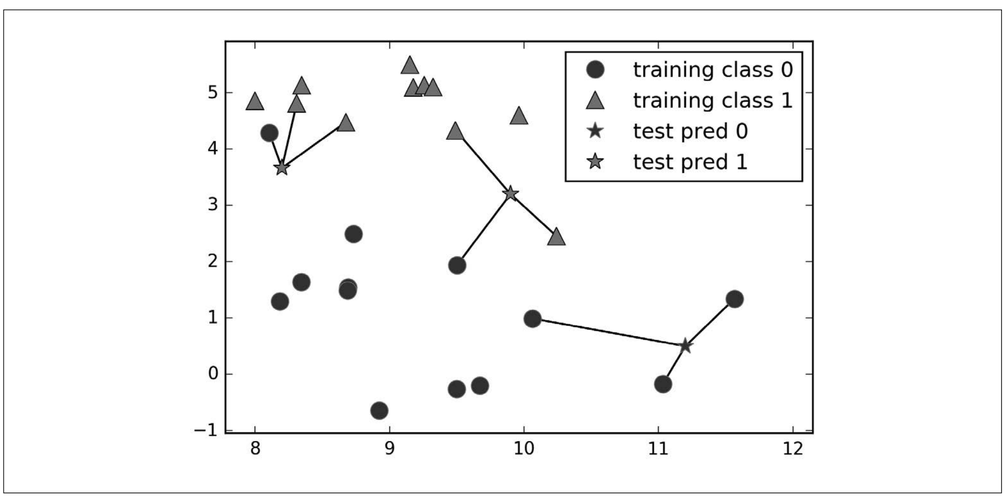


图 2-5：3 近邻模型对 forge 数据集的预测结果

和上面一样，预测结果可以从五角星的颜色看出。你可以发现，左上角新数据点的预测结果与只用一个邻居时的预测结果不同。

虽然这张图对应的是一个二分类问题，但方法同样适用于多分类的数据集。对于多分类问题，我们数一数每个类别分别有多少个邻居，然后将最常见的类别作为预测结果。

现在看一下如何通过 `scikit-learn` 来应用 k 近邻算法。首先，正如第 1 章所述，将数据分为训练集和测试集，以便评估泛化性能：

In[12]:

```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

然后，导入类并将其实例化。这时可以设定参数，比如邻居的个数。这里我们将其设为 3：

In[13]:

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

现在，利用训练集对这个分类器进行拟合。对于 `KNeighborsClassifier` 来说就是保存数据集，以便在预测时计算与邻居之间的距离：

In[14]:

```
clf.fit(X_train, y_train)
```

调用 `predict` 方法来对测试数据进行预测。对于测试集中的每个数据点，都要计算它在训练集的最近邻，然后找出其中出现次数最多的类别：

In[15]:

```
print("Test set predictions: {}".format(clf.predict(X_test)))
```

Out[15]:

```
Test set predictions: [1 0 1 0 1 0 0]
```

为了评估模型的泛化能力好坏，我们可以对测试数据和测试标签调用 `score` 方法：

In[16]:

```
print("Test set accuracy: {:.2f}".format(clf.score(X_test, y_test)))
```

Out[16]:

```
Test set accuracy: 0.86
```

可以看到，我们的模型精度约为 86%，也就是说，在测试数据集中，模型对其中 86% 的样本预测的类别都是正确的。

2. 分析`KNeighborsClassifier`

对于二维数据集，我们还可以在 xy 平面上画出所有可能的测试点的预测结果。我们根据平面中每个点所属的类别对平面进行着色。这样可以查看**决策边界**（decision boundary），即算法对类别 0 和类别 1 的分界线。

下列代码分别将 1 个、3 个和 9 个邻居三种情况的决策边界可视化，见图 2-6：

In[17]:

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
    # fit方法返回对象本身，所以我们可以将实例化和拟合放在一行代码中
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{} neighbor(s)".format(n_neighbors))
    ax.set_xlabel("feature 0")
    ax.set_ylabel("feature 1")
axes[0].legend(loc=3)
```

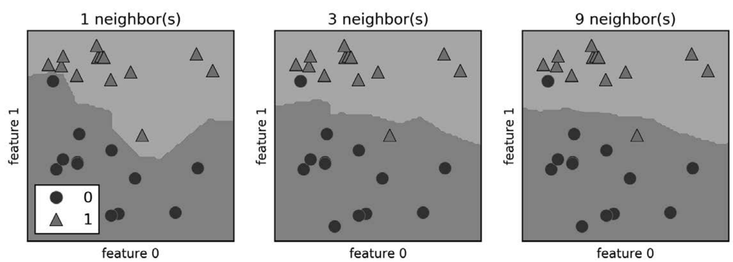


图 2-6：不同 n_neighbors 值的 k 近邻模型的决策边界

从左图可以看出，使用单一邻居绘制的决策边界紧跟着训练数据。随着邻居个数越来越多，决策边界也越来越平滑。更平滑的边界对应更简单的模型。换句话说，使用更少的邻居对应更高的模型复杂度（如图 2-1 右侧所示），而使用更多的邻居对应更低的模型复杂度（如图 2-1 左侧所示）。假如考虑极端情况，即邻居个数等于训练集中所有数据点的个数，那么每个测试点的邻居都完全相同（即所有训练点），所有预测结果也完全相同（即训练集中出现次数最多的类别）。

我们来研究一下能否证实之前讨论过的模型复杂度和泛化能力之间的关系。我们将在现实世界的乳腺癌数据集上进行研究。先将数据集分成训练集和测试集，然后用不同的邻居个数对训练集和测试集的性能进行评估。输出结果见图 2-7：

In[18]:

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []
# n_neighbors取值从1到10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
```

```
    # 构建模型
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # 记录训练集精度
    training_accuracy.append(clf.score(X_train, y_train))
    # 记录泛化精度
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
```

图像的 x 轴是 n_neighbors，y 轴是训练集精度和测试集精度。虽然现实世界的图像很少有非常平滑的，但我们仍可以看出过拟合与欠拟合的一些特征（注意，由于更少的邻居对应更复杂的模型，所以此图相对于图 2-1 做了水平翻转）。仅考虑单一近邻时，训练集上的预测结果十分完美。但随着邻居个数的增多，模型变得更简单，训练集精度也随之下降。单一邻居时的测试集精度比使用更多邻居时要低，这表示单一近邻的模型过于复杂。与之相反，当考虑 10 个邻居时，模型又过于简单，性能甚至变得更差。最佳性能在中间的某处，邻居个数大约为 6。不过最好记住这张图的坐标轴刻度。最差的性能约为 88% 的精度，这个结果仍然可以接受。

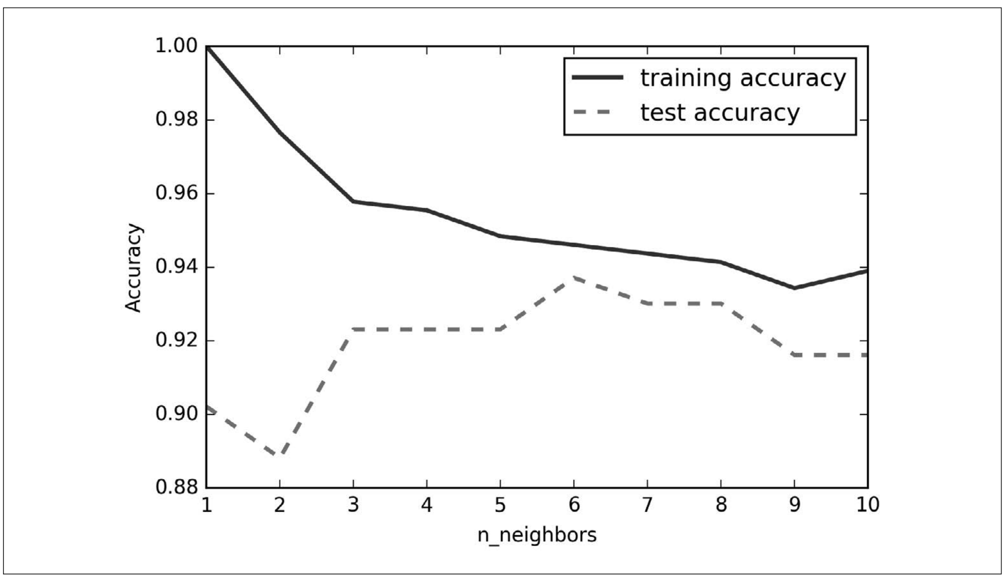


图 2-7：以 n_neighbors 为自变量，对比训练集精度和测试集精度

3. k近邻回归

k 近邻算法还可以用于回归。我们还是先从单一近邻开始，这次使用 wave 数据集。我们添加了 3 个测试数据点，在 x 轴上用绿色五角星表示。利用单一邻居的预测结果就是最近邻的目标值。在图 2-8 中用蓝色五角星表示：

In[19]:

```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

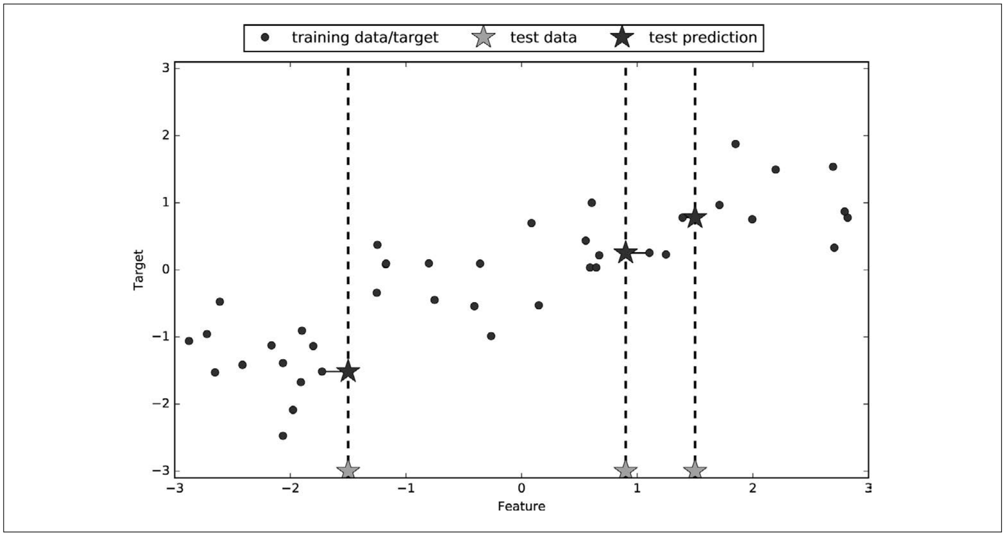


图 2-8：单一近邻回归对 wave 数据集的预测结果

同样，也可以用多个近邻进行回归。在使用多个近邻时，预测结果为这些邻居的平均值（图 2-9）：

In[20]:

```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```

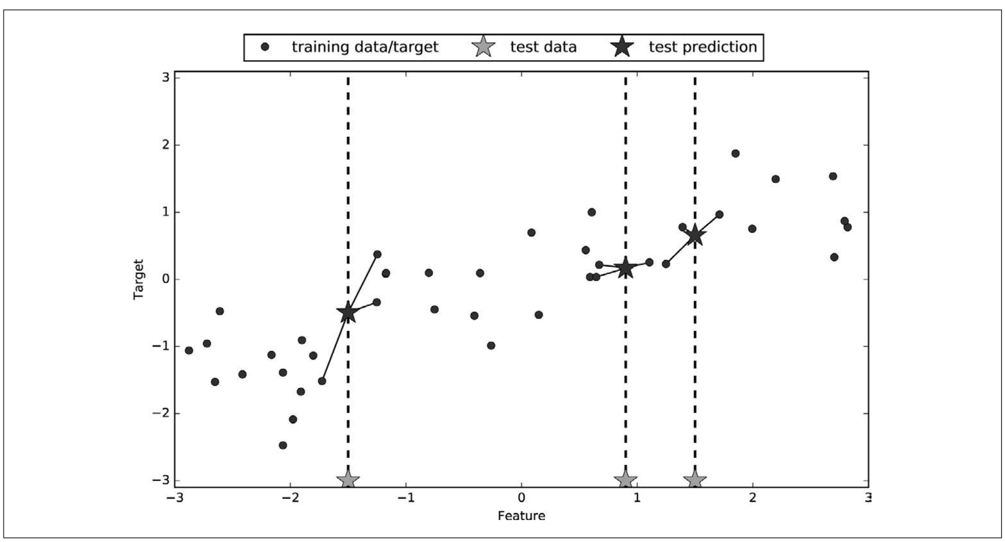


图 2-9：3 个近邻回归对 wave 数据集的预测结果

用于回归的 k 近邻算法在 scikit-learn 的 KNeighborsRegressor 类中实现。其用法与 KNeighborsClassifier 类似：

In[21]:

```
from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make_wave(n_samples=40)

# 将wave数据集分为训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# 模型实例化，并将邻居个数设为3
reg = KNeighborsRegressor(n_neighbors=3)
# 利用训练数据和训练目标值来拟合模型
reg.fit(X_train, y_train)
```

现在可以对测试集进行预测：

In[22]:

```
print("Test set predictions:\n{}".format(reg.predict(X_test)))
```

Out[22]:

```
Test set predictions:
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912 -0.447 -1.139]
```

我们还可以用 score 方法来评估模型，对于回归问题，这一方法返回的是 R^2 分数。R^2 分数也叫作决定系数，是回归模型预测的优度度量，位于 0 到 1 之间。R^2 等于 1 对应完美预测，R^2 等于 0 对应常数模型，即总是预测训练集响应（y_train）的平均值：

In[23]:

```
print("Test set R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

Out[23]:

```
Test set R^2: 0.83
```

这里的分数是 0.83，表示模型的拟合相对较好。

4. 分析KNeighborsRegressor

对于我们的一维数据集，可以查看所有特征取值对应的预测结果（图 2-10）。为了便于绘图，我们创建一个由许多点组成的测试数据集：

In[24]:

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
# 创建1000个数据点，在-3和3之间均匀分布
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
    # 利用1个、3个或9个邻居分别进行预测
    reg = KNeighborsRegressor(n_neighbors=n_neighbors)
    reg.fit(X_train, y_train)
    ax.plot(line, reg.predict(line))
    ax.plot(X_train, y_train, '^', c=mglearn.cm2(0), markersize=8)
    ax.plot(X_test, y_test, 'v', c=mglearn.cm2(1), markersize=8)
```

```
        ax.set_title(
            "{} neighbor(s)\n train score: {:.2f} test score: {:.2f}".format(
                n_neighbors, reg.score(X_train, y_train),
                reg.score(X_test, y_test)))
        ax.set_xlabel("Feature")
        ax.set_ylabel("Target")
    axes[0].legend(["Model predictions", "Training data/target",
                    "Test data/target"], loc="best")
```

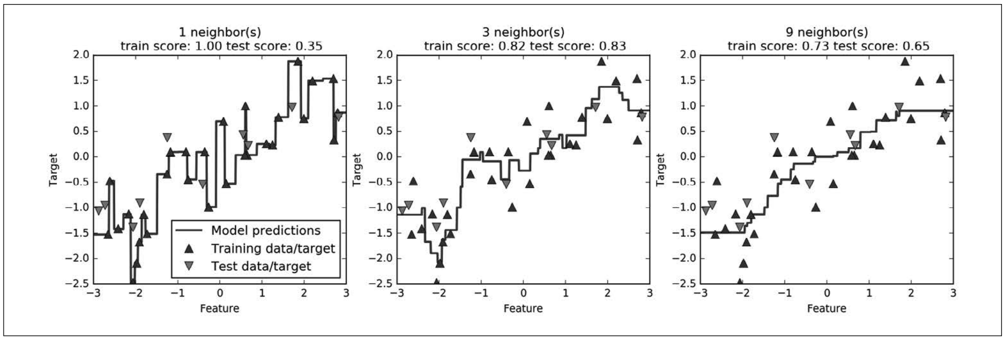


图 2-10：不同 n_neighbors 值的 k 近邻回归的预测结果对比

从图中可以看出，仅使用单一邻居，训练集中的每个点都对预测结果有显著影响，预测结果的图像经过所有数据点。这导致预测结果非常不稳定。考虑更多的邻居之后，预测结果变得更加平滑，但对训练数据的拟合也不好。

5. 优点、缺点和参数

一般来说，KNeighbors 分类器有 2 个重要参数：邻居个数与数据点之间距离的度量方法。在实践中，使用较小的邻居个数（比如 3 个或 5 个）往往可以得到比较好的结果，但你应该调节这个参数。选择合适的距离度量方法超出了本书的范围。默认使用欧式距离，它在许多情况下的效果都很好。

k-NN 的优点之一就是模型很容易理解，通常不需要过多调节就可以得到不错的性能。在考虑使用更高级的技术之前，尝试此算法是一种很好的基准方法。构建最近邻模型的速度通常很快，但如果训练集很大（特征数很多或者样本数很大），预测速度可能会比较慢。使用 k-NN 算法时，对数据进行预处理是很重要的（见第 3 章）。这一算法对于有很多特征（几百或更多）的数据集往往效果不好，对于大多数特征的大多数取值都为 0 的数据集（所谓的**稀疏数据集**）来说，这一算法的效果尤其不好。

虽然 k 近邻算法很容易理解，但由于预测速度慢且不能处理具有很多特征的数据集，所以在实践中往往不会用到。下面介绍的这种方法就没有这两个缺点。

2.3.3　线性模型

线性模型是在实践中广泛使用的一类模型，几十年来被广泛研究，它可以追溯到一百多年前。线性模型利用输入特征的**线性函数**（linear function）进行预测，稍后会对此进行解释。

1. 用于回归的线性模型

对于回归问题，线性模型预测的一般公式如下：

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \cdots + w[p] * x[p] + b$$

这里 $x[0]$ 到 $x[p]$ 表示单个数据点的特征（本例中特征个数为 p+1），w 和 b 是学习模型的参数，$\hat{y}$ 是模型的预测结果。对于单一特征的数据集，公式如下：

$$\hat{y} = w[0] * x[0] + b$$

你可能还记得，这就是高中数学里的直线方程。这里 $w[0]$ 是斜率，b 是 y 轴偏移。对于有更多特征的数据集，w 包含沿每个特征坐标轴的斜率。或者，你也可以将预测的响应值看作输入特征的加权求和，权重由 w 的元素给出（可以取负值）。

下列代码可以在一维 wave 数据集上学习参数 $w[0]$ 和 b：

In[25]:

```
mglearn.plots.plot_linear_regression_wave()
```

Out[25]:

```
w[0]: 0.393906  b: -0.031804
```

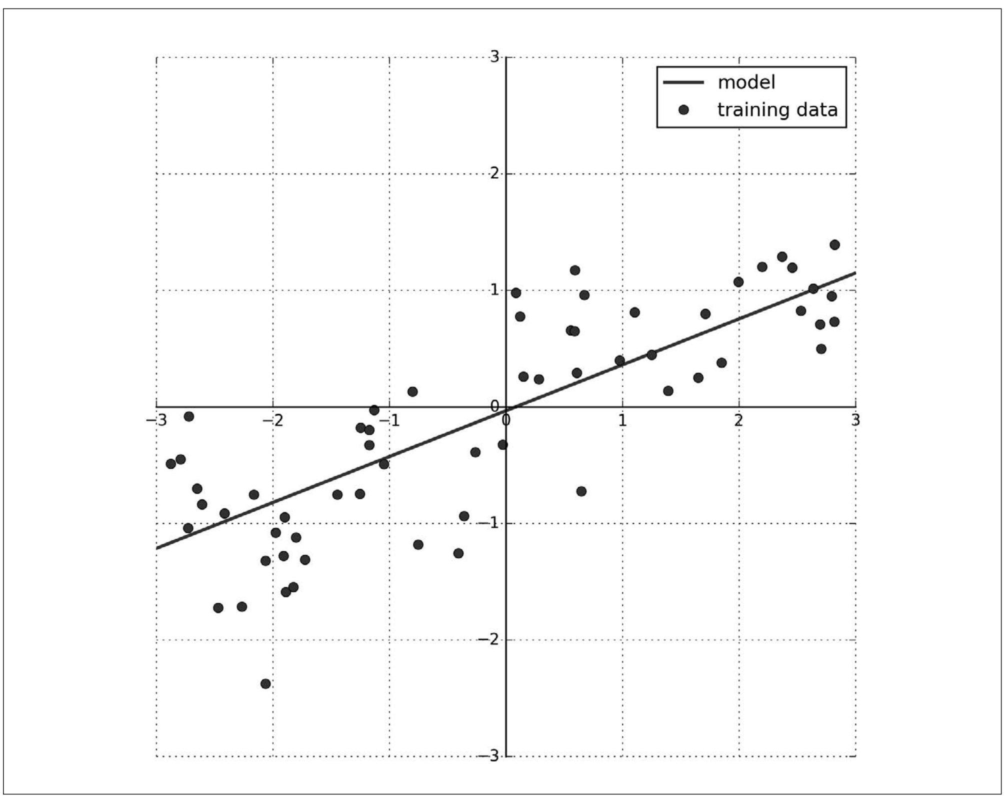


图 2-11：线性模型对 wave 数据集的预测结果

我们在图中添加了坐标网格，便于理解直线的含义。从 `w[0]` 可以看出，斜率应该在 0.4 左右，在图像中也可以直观地确认这一点。截距是指预测直线与 y 轴的交点：比 0 略小，也可以在图像中确认。

用于回归的线性模型可以表示为这样的回归模型：对单一特征的预测结果是一条直线，两个特征时是一个平面，或者在更高维度（即更多特征）时是一个超平面。

如果将直线的预测结果与图 2-10 中 `KNeighborsRegressor` 的预测结果进行比较，你会发现直线的预测能力非常受限。似乎数据的所有细节都丢失了。从某种意义上来说，这种说法是正确的。假设目标 y 是特征的线性组合，这是一个非常强的（也有点不现实的）假设。但观察一维数据得出的观点有些片面。对于有多个特征的数据集而言，线性模型可以非常强大。特别地，如果特征数量大于训练数据点的数量，任何目标 y 都可以（在训练集上）用线性函数完美拟合[6]。

有许多不同的线性回归模型。这些模型之间的区别在于如何从训练数据中学习参数 w 和 b，以及如何控制模型复杂度。下面介绍最常见的线性回归模型。

2. 线性回归（又名普通最小二乘法）

线性回归，或者**普通最小二乘法**（ordinary least squares，OLS），是回归问题最简单也最经典的线性方法。线性回归寻找参数 w 和 b，使得对训练集的预测值与真实的回归目标值 y 之间的**均方误差**最小。均方误差（mean squared error）是预测值与真实值之差的平方和除以样本数。线性回归没有参数，这是一个优点，但也因此无法控制模型的复杂度。

下列代码可以生成图 2-11 中的模型：

In[26]:

```
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)
```

“斜率”参数（w，也叫作权重或**系数**）被保存在 `coef_` 属性中，而偏移或**截距**（b）被保存在 `intercept_` 属性中：

In[27]:

```
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
```

Out[27]:

```
lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746
```

> 你可能注意到了 `coef_` 和 `intercept_` 结尾处奇怪的下划线。`scikit-learn` 总是将从训练数据中得出的值保存在以下划线结尾的属性中。这是为了将其与用户设置的参数区分开。

注 6：如果你懂一点线性代数的话，很容易理解这一点。

intercept_ 属性是一个浮点数，而 coef_ 属性是一个 NumPy 数组，每个元素对应一个输入特征。由于 wave 数据集中只有一个输入特征，所以 lr.coef_ 中只有一个元素。

我们来看一下训练集和测试集的性能：

In[28]:

```
print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

Out[28]:

```
Training set score: 0.67
Test set score: 0.66
```

R^2 约为 0.66，这个结果不是很好，但我们可以看到，训练集和测试集上的分数非常接近。这说明可能存在欠拟合，而不是过拟合。对于这个一维数据集来说，过拟合的风险很小，因为模型非常简单（或受限）。然而，对于更高维的数据集（即有大量特征的数据集），线性模型将变得更加强大，过拟合的可能性也会变大。我们来看一下 LinearRegression 在更复杂的数据集上的表现，比如波士顿房价数据集。记住，这个数据集有 506 个样本和 105 个导出特征。首先，加载数据集并将其分为训练集和测试集。然后像前面一样构建线性回归模型：

In[29]:

```
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

比较一下训练集和测试集的分数就可以发现，我们在训练集上的预测非常准确，但测试集上的 R^2 要低很多：

In[30]:

```
print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

Out[30]:

```
Training set score: 0.95
Test set score: 0.61
```

训练集和测试集之间的性能差异是过拟合的明显标志，因此我们应该试图找到一个可以控制复杂度的模型。标准线性回归最常用的替代方法之一就是**岭回归**（ridge regression），下面来看一下。

3. 岭回归

岭回归也是一种用于回归的线性模型，因此它的预测公式与普通最小二乘法相同。但在岭回归中，对系数（w）的选择不仅要在训练数据上得到好的预测结果，而且还要拟合附加约束。我们还希望系数尽量小。换句话说，w 的所有元素都应接近于 0。直观上来看，这意味着每个特征对输出的影响应尽可能小（即斜率很小），同时仍给出很好的预测结果。这种约束是所谓**正则化**（regularization）的一个例子。正则化是指对模型做显式约束，以

避免过拟合。岭回归用到的这种被称为 L2 正则化。[7]

岭回归在 linear_model.Ridge 中实现。来看一下它对扩展的波士顿房价数据集的效果如何：

In[31]:

```
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge.score(X_test, y_test)))
```

Out[31]:

```
Training set score: 0.89
Test set score: 0.75
```

可以看出，Ridge 在训练集上的分数要**低**于 LinearRegression，但在测试集上的分数**更高**。这和我们的预期一致。线性回归对数据存在过拟合。Ridge 是一种约束更强的模型，所以更不容易过拟合。复杂度更小的模型意味着在训练集上的性能更差，但泛化性能更好。由于我们只对泛化性能感兴趣，所以应该选择 Ridge 模型而不是 LinearRegression 模型。

Ridge 模型在模型的简单性（系数都接近于 0）与训练集性能之间做出权衡。简单性和训练集性能二者对于模型的重要程度可以由用户通过设置 alpha 参数来指定。在前面的例子中，我们用的是默认参数 alpha=1.0。但没有理由认为这会给出最佳权衡。alpha 的最佳设定值取决于用到的具体数据集。增大 alpha 会使得系数更加趋向于 0，从而降低训练集性能，但可能会提高泛化性能。例如：

In[32]:

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge10.score(X_test, y_test)))
```

Out[32]:

```
Training set score: 0.79
Test set score: 0.64
```

减小 alpha 可以让系数受到的限制更小，即在图 2-1 中向右移动。对于非常小的 alpha 值，系数几乎没有受到限制，我们得到一个与 LinearRegression 类似的模型：

In[33]:

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge01.score(X_test, y_test)))
```

Out[33]:

```
Training set score: 0.93
Test set score: 0.77
```

注 7：从数学的观点来看，Ridge 惩罚了系数的 L2 范数或 w 的欧式长度。

这里 alpha=0.1 似乎效果不错。我们可以尝试进一步减小 alpha 以提高泛化性能。现在，注意参数 alpha 与图 2-1 中的模型复杂度的对应关系。第 5 章将会讨论选择参数的正确方法。

我们还可以查看 alpha 取不同值时模型的 coef_ 属性，从而更加定性地理解 alpha 参数是如何改变模型的。更大的 alpha 表示约束更强的模型，所以我们预计大 alpha 对应的 coef_ 元素比小 alpha 对应的 coef_ 元素要小。这一点可以在图 2-12 中得到证实：

In[34]:

```
plt.plot(ridge.coef_, 's', label="Ridge alpha=1")
plt.plot(ridge10.coef_, '^', label="Ridge alpha=10")
plt.plot(ridge01.coef_, 'v', label="Ridge alpha=0.1")

plt.plot(lr.coef_, 'o', label="LinearRegression")
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```

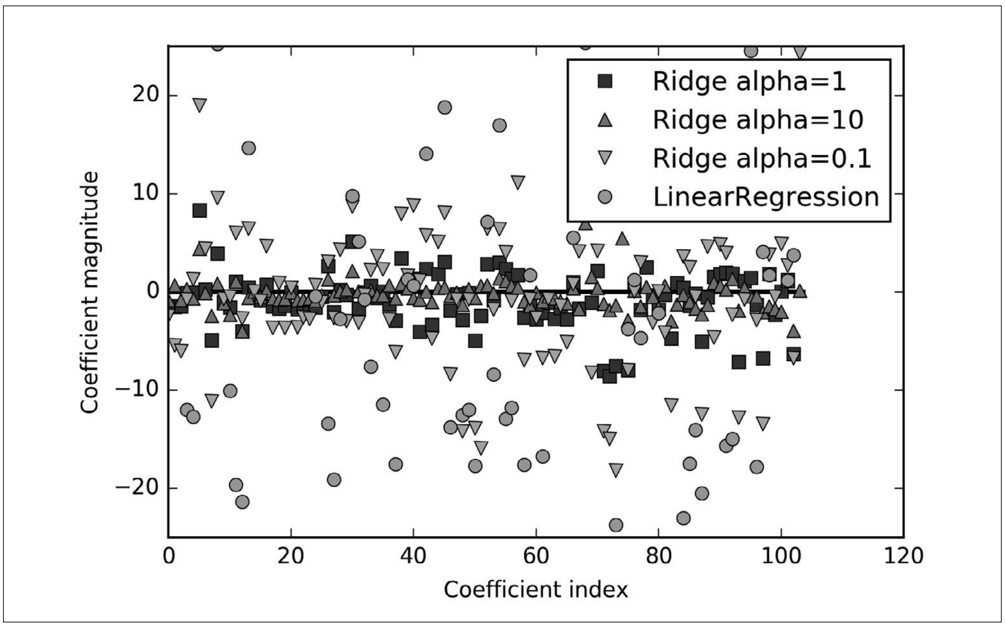


图 2-12：不同 alpha 值的岭回归与线性回归的系数比较

这里 x 轴对应 coef_ 的元素：x=0 对应第一个特征的系数，x=1 对应第二个特征的系数，以此类推，一直到 x=100。y 轴表示该系数的具体数值。这里需要记住的是，对于 alpha=10，系数大多在 −3 和 3 之间。对于 alpha=1 的 Ridge 模型，系数要稍大一点。对于 alpha=0.1，点的范围更大。对于没有做正则化的线性回归（即 alpha=0），点的范围很大，许多点都超出了图像的范围。

还有一种方法可以用来理解正则化的影响，就是固定 alpha 值，但改变训练数据量。对于

图 2-13 来说，我们对波士顿房价数据集做二次抽样，并在数据量逐渐增加的子数据集上分别对 `LinearRegression` 和 `Ridge(alpha=1)` 两个模型进行评估（将模型性能作为数据集大小的函数进行绘图，这样的图像叫作**学习曲线**）：

In[35]:

```
mglearn.plots.plot_ridge_n_samples()
```

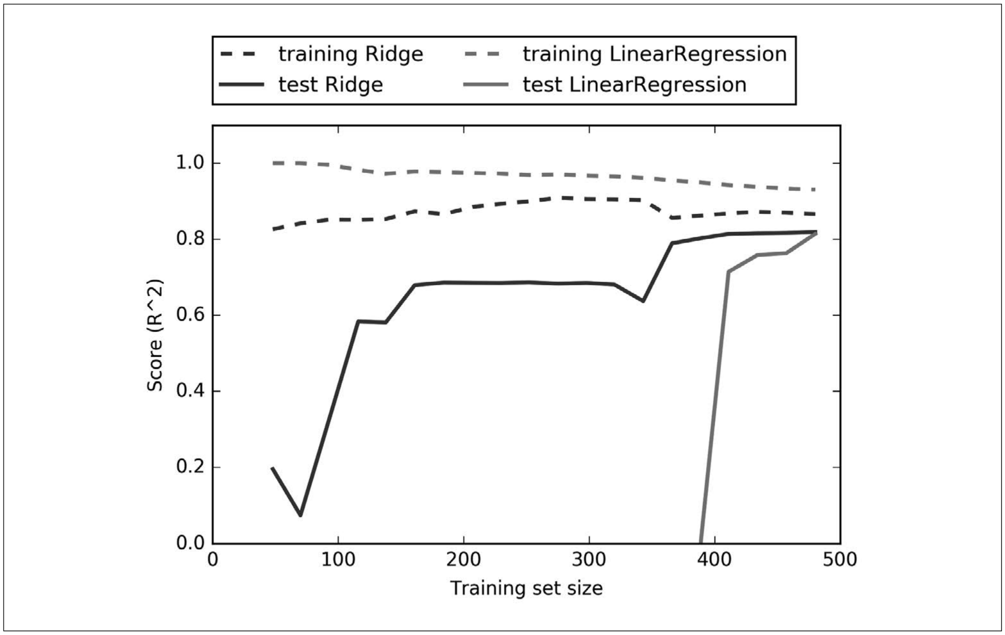


图 2-13：岭回归和线性回归在波士顿房价数据集上的学习曲线

正如所预计的那样，无论是岭回归还是线性回归，所有数据集大小对应的训练分数都要高于测试分数。由于岭回归是正则化的，因此它的训练分数要整体低于线性回归的训练分数。但岭回归的测试分数要更高，特别是对较小的子数据集。如果少于 400 个数据点，线性回归学不到任何内容。随着模型可用的数据越来越多，两个模型的性能都在提升，最终线性回归的性能追上了岭回归。这里要记住的是，如果有足够多的训练数据，正则化变得不那么重要，并且岭回归和线性回归将具有相同的性能（在这个例子中，二者相同恰好发生在整个数据集的情况下，这只是一个巧合）。图 2-13 中还有一个有趣之处，就是线性回归的训练性能在下降。如果添加更多数据，模型将更加难以过拟合或记住所有的数据。

4. lasso

除了 `Ridge`，还有一种正则化的线性回归是 `Lasso`。与岭回归相同，使用 lasso 也是约束系数使其接近于 0，但用到的方法不同，叫作 L1 正则化。[8] L1 正则化的结果是，使用 lasso 时某些系数**刚好为 0**。这说明某些特征被模型完全忽略。这可以看作是一种自动化的特征选择。某些系数刚好为 0，这样模型更容易解释，也可以呈现模型最重要的特征。

注 8：lasso 惩罚系数向量的 L1 范数，换句话说，系数的绝对值之和。

我们将 lasso 应用在扩展的波士顿房价数据集上：

In[36]:

```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))
```

Out[36]:

```
Training set score: 0.29
Test set score: 0.21
Number of features used: 4
```

如你所见，`Lasso` 在训练集与测试集上的表现都很差。这表示存在欠拟合，我们发现模型只用到了 105 个特征中的 4 个。与 `Ridge` 类似，`Lasso` 也有一个正则化参数 `alpha`，可以控制系数趋向于 0 的强度。在上一个例子中，我们用的是默认值 `alpha=1.0`。为了降低欠拟合，我们尝试减小 `alpha`。这么做的同时，我们还需要增加 `max_iter` 的值（运行迭代的最大次数）：

In[37]:

```
# 我们增大max_iter的值，否则模型会警告我们，说应该增大max_iter
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso001.coef_ != 0)))
```

Out[37]:

```
Training set score: 0.90
Test set score: 0.77
Number of features used: 33
```

`alpha` 值变小，我们可以拟合一个更复杂的模型，在训练集和测试集上的表现也更好。模型性能比使用 `Ridge` 时略好一点，而且我们只用到了 105 个特征中的 33 个。这样模型可能更容易理解。

但如果把 `alpha` 设得太小，那么就会消除正则化的效果，并出现过拟合，得到与 `LinearRegression` 类似的结果：

In[38]:

```
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso00001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso00001.coef_ != 0)))
```

Out[38]:

```
Training set score: 0.95
Test set score: 0.64
Number of features used: 94
```

再次像图 2-12 那样对不同模型的系数进行作图，见图 2-14：

In[39]:

```
plt.plot(lasso.coef_, 's', label="Lasso alpha=1")
plt.plot(lasso001.coef_, '^', label="Lasso alpha=0.01")
plt.plot(lasso00001.coef_, 'v', label="Lasso alpha=0.0001")

plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1")
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
```

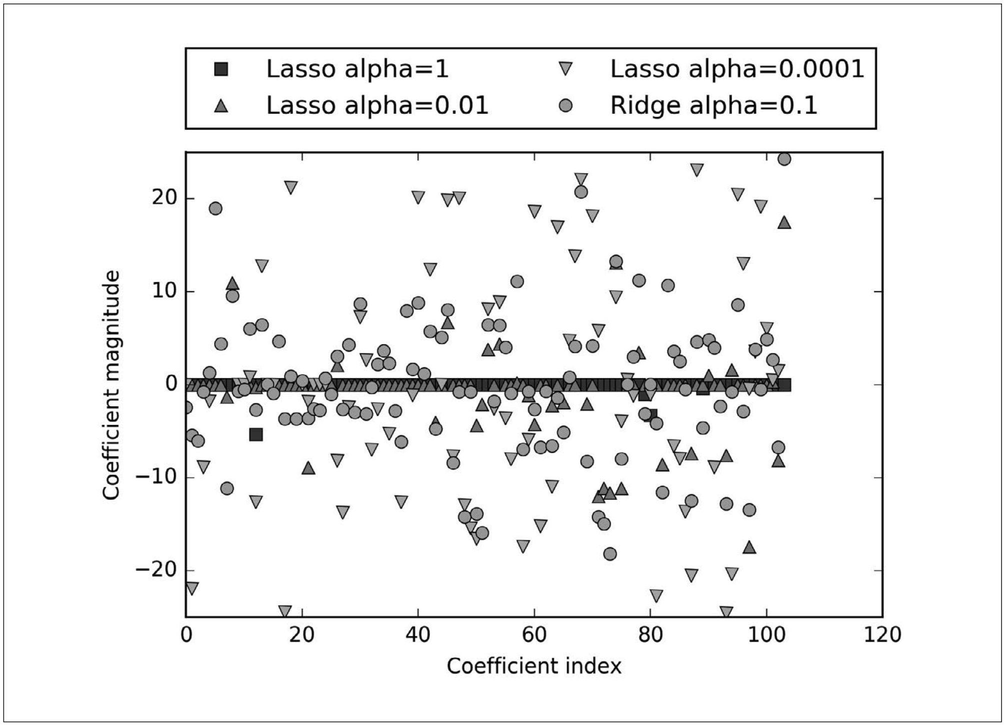


图 2-14：不同 alpha 值的 lasso 回归与岭回归的系数比较

在 alpha=1 时，我们发现不仅大部分系数都是 0（我们已经知道这一点），而且其他系数也都很小。将 alpha 减小至 0.01，我们得到图中向上的三角形，大部分特征等于 0。alpha=0.0001 时，我们得到正则化很弱的模型，大部分系数都不为 0，并且还很大。为了便于比较，图中用圆形表示 Ridge 的最佳结果。alpha=0.1 的 Ridge 模型的预测性能与 alpha=0.01 的 Lasso 模型类似，但 Ridge 模型的所有系数都不为 0。

在实践中，在两个模型中一般首选岭回归。但如果特征很多，你认为只有其中几个是重要的，那么选择 Lasso 可能更好。同样，如果你想要一个容易解释的模型，Lasso 可以给出更容易理解的模型，因为它只选择了一部分输入特征。scikit-learn 还提供了 ElasticNet 类，结合了 Lasso 和 Ridge 的惩罚项。在实践中，这种结合的效果最好，不过代价是要调节两个参数：一个用于 L1 正则化，一个用于 L2 正则化。

5. 用于分类的线性模型

线性模型也广泛应用于分类问题。我们首先来看二分类。这时可以利用下面的公式进行预测：

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \cdots + w[p] * x[p] + b > 0$$

这个公式看起来与线性回归的公式非常相似，但我们没有返回特征的加权求和，而是为预测设置了阈值（0）。如果函数值小于 0，我们就预测类别 -1；如果函数值大于 0，我们就预测类别 +1。对于所有用于分类的线性模型，这个预测规则都是通用的。同样，有很多种不同的方法来找出系数（w）和截距（b）。

对于用于回归的线性模型，输出 $\hat{y}$ 是特征的线性函数，是直线、平面或超平面（对于更高维的数据集）。对于用于分类的线性模型，**决策边界**是输入的线性函数。换句话说，（二元）线性分类器是利用直线、平面或超平面来分开两个类别的分类器。本节我们将看到这方面的例子。

学习线性模型有很多种算法。这些算法的区别在于以下两点：

- 系数和截距的特定组合对训练数据拟合好坏的度量方法；
- 是否使用正则化，以及使用哪种正则化方法。

不同的算法使用不同的方法来度量“对训练集拟合好坏”。由于数学上的技术原因，不可能调节 w 和 b 使得算法产生的误分类数量最少。对于我们的目的，以及对于许多应用而言，上面第一点（称为**损失函数**）的选择并不重要。

最常见的两种线性分类算法是 **Logistic 回归**（logistic regression）和**线性支持向量机**（linear support vector machine，线性 SVM），前者在 `linear_model.LogisticRegression` 中实现，后者在 `svm.LinearSVC`（SVC 代表支持向量分类器）中实现。虽然 `LogisticRegression` 的名字中含有回归（regression），但它是一种分类算法，并不是回归算法，不应与 `LinearRegression` 混淆。

我们可以将 `LogisticRegression` 和 `LinearSVC` 模型应用到 `forge` 数据集上，并将线性模型找到的决策边界可视化（图 2-15）：

In[40]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5,
                                    ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
axes[0].legend()
```

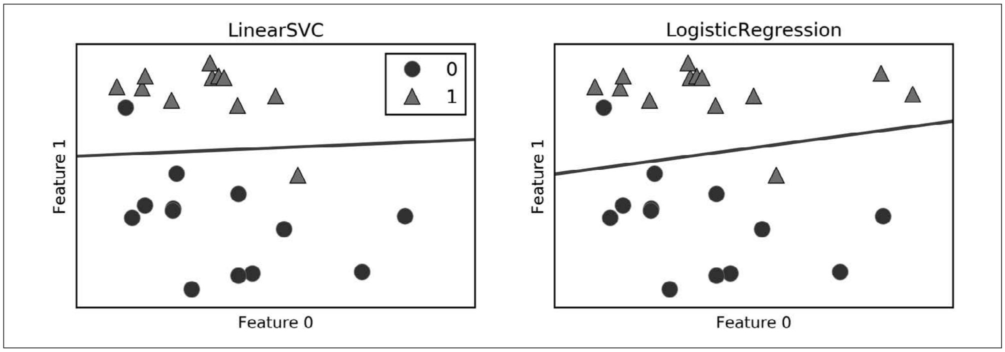


图 2-15：线性 SVM 和 Logistic 回归在 forge 数据集上的决策边界（均为默认参数）

在这张图中，forge 数据集的第一个特征位于 x 轴，第二个特征位于 y 轴，与前面相同。图中分别展示了 LinearSVC 和 LogisticRegression 得到的决策边界，都是直线，将顶部归为类别 1 的区域和底部归为类别 0 的区域分开了。换句话说，对于每个分类器而言，位于黑线上方的新数据点都会被划为类别 1，而在黑线下方的点都会被划为类别 0。

两个模型得到了相似的决策边界。注意，两个模型中都有两个点的分类是错误的。两个模型都默认使用 L2 正则化，就像 Ridge 对回归所做的那样。

对于 LogisticRegression 和 LinearSVC，决定正则化强度的权衡参数叫作 C。C 值越大，对应的正则化**越弱**。换句话说，如果参数 C 值较大，那么 LogisticRegression 和 LinearSVC 将尽可能将训练集拟合到最好，而如果 C 值较小，那么模型更强调使系数向量（w）接近于 0。

参数 C 的作用还有另一个有趣之处。较小的 C 值可以让算法尽量适应“大多数”数据点，而较大的 C 值更强调每个数据点都分类正确的重要性。下面是使用 LinearSVC 的图示（图 2-16）：

In[41]:

```
mglearn.plots.plot_linear_svc_regularization()
```

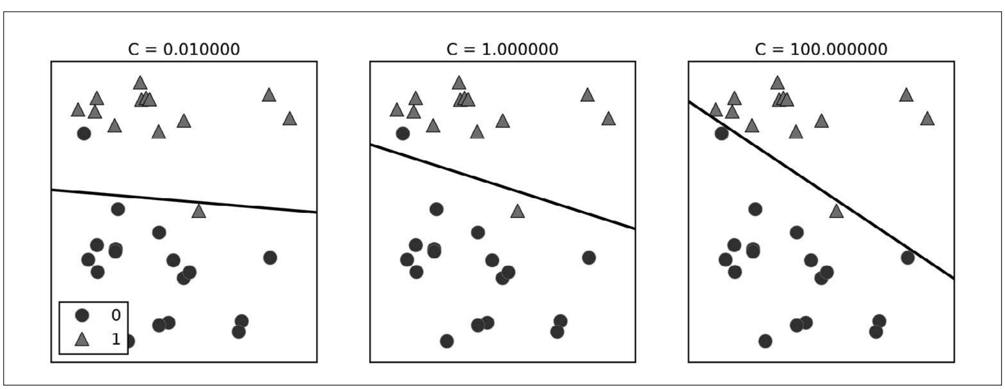


图 2-16：不同 C 值的线性 SVM 在 forge 数据集上的决策边界

在左侧的图中，C 值很小，对应强正则化。大部分属于类别 0 的点都位于底部，大部分属于类别 1 的点都位于顶部。强正则化的模型会选择一条相对水平的线，有两个点分类错误。在中间的图中，C 值稍大，模型更关注两个分类错误的样本，使决策边界的斜率变大。最后，在右侧的图中，模型的 C 值非常大，使得决策边界的斜率也很大，现在模型对类别 0 中所有点的分类都是正确的。类别 1 中仍有一个点分类错误，这是因为对这个数据集来说，不可能用一条直线将所有点都分类正确。右侧图中的模型尽量使所有点的分类都正确，但可能无法掌握类别的整体分布。换句话说，这个模型很可能过拟合。

与回归的情况类似，用于分类的线性模型在低维空间中看起来可能非常受限，决策边界只能是直线或平面。同样，在高维空间中，用于分类的线性模型变得非常强大，当考虑更多特征时，避免过拟合变得越来越重要。

我们在乳腺癌数据集上详细分析 LogisticRegression：

In[42]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))
```

Out[42]:

```
Training set score: 0.953
Test set score: 0.958
```

C=1 的默认值给出了相当好的性能，在训练集和测试集上都达到 95% 的精度。但由于训练集和测试集的性能非常接近，所以模型很可能是欠拟合的。我们尝试增大 C 来拟合一个更灵活的模型：

In[43]:

```
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg100.score(X_test, y_test)))
```

Out[43]:

```
Training set score: 0.972
Test set score: 0.965
```

使用 C=100 可以得到更高的训练集精度，也得到了稍高的测试集精度，这也证实了我们的直觉，即更复杂的模型应该性能更好。

我们还可以研究使用正则化更强的模型时会发生什么。设置 C=0.01：

In[44]:

```
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg001.score(X_test, y_test)))
```

Out[44]:

```
Training set score: 0.934
Test set score: 0.930
```

正如我们所料，在图 2-1 中将已经欠拟合的模型继续向左移动，训练集和测试集的精度都比采用默认参数时更小。

最后，来看一下正则化参数 C 取三个不同的值时模型学到的系数（图 2-17）：

In[45]:

```
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.legend()
```

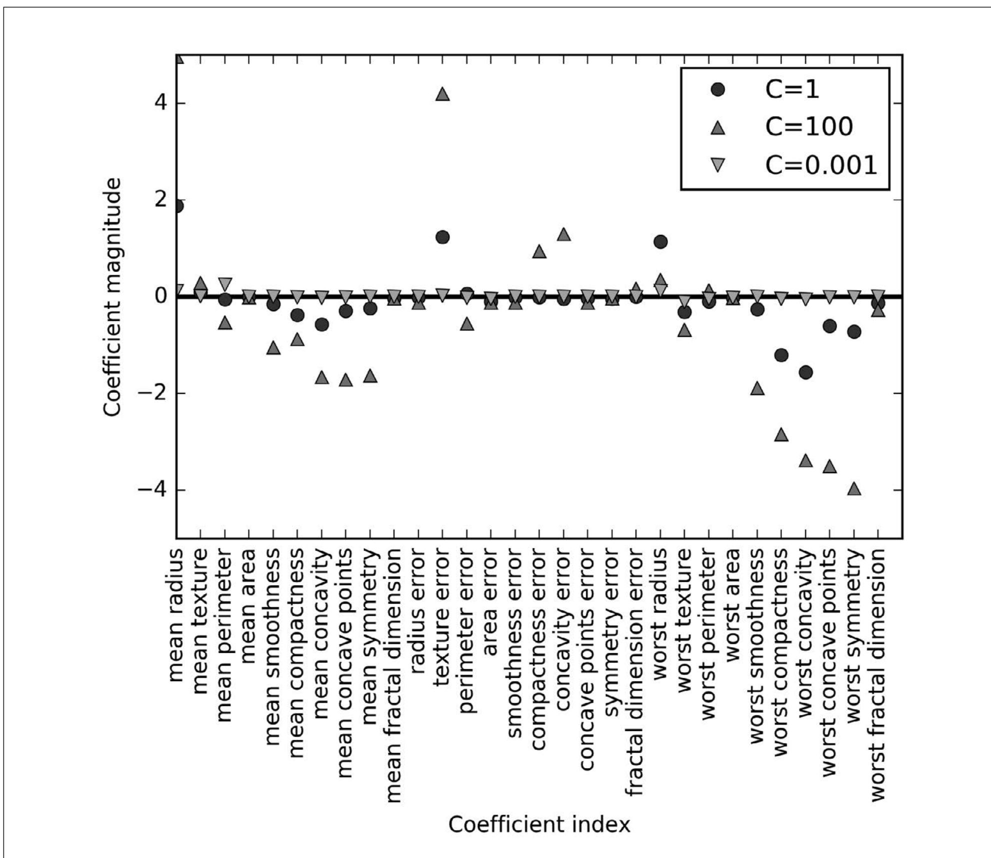


图 2-17：不同 C 值的 Logistic 回归在乳腺癌数据集上学到的系数

由于 `LogisticRegression` 默认应用 L2 正则化，所以其结果与图 2-12 中 `Ridge` 的结果类似。更强的正则化使得系数更趋向于 0，但系数永远不会正好等于 0。进一步观察图像，还可以在第 3 个系数那里发现有趣之处，这个系数是“平均周长”（mean perimeter）。`C=100` 和 `C=1` 时，这个系数为负，而 `C=0.001` 时这个系数为正，其绝对值比 `C=1` 时还要大。在解释这样的模型时，人们可能会认为，系数可以告诉我们某个特征与哪个类别有关。例如，人们可能会认为高“纹理错误”（texture error）特征与“恶性”样本有关。但“平均周长”系数的正负号发生变化，说明较大的“平均周长”可以被当作“良性”的指标或“恶性”的指标，具体取决于我们考虑的是哪个模型。这也说明，对线性模型系数的解释应该始终持保留态度。

如果想要一个可解释性更强的模型，使用 L1 正则化可能更好，因为它约束模型只使用少数几个特征。下面是使用 L1 正则化的系数图像和分类精度（图 2-18）。

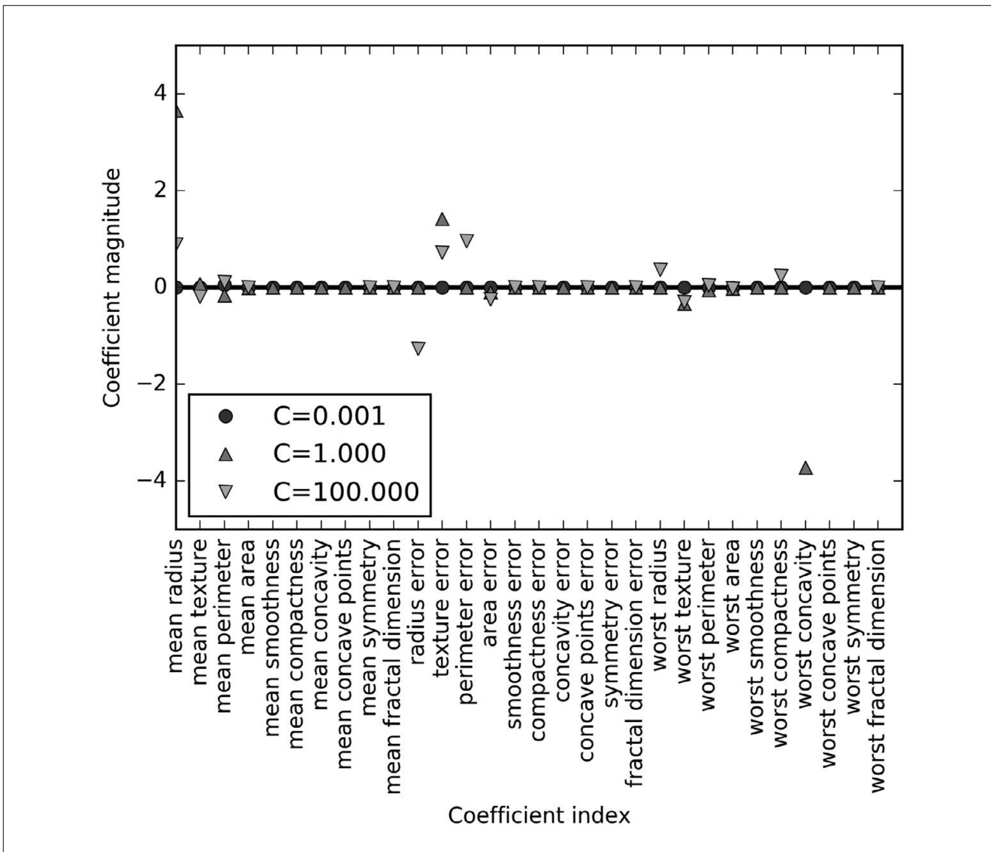


图 2-18：对于不同的 C 值，L1 惩罚的 Logistic 回归在乳腺癌数据集上学到的系数

In[46]:

```
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
    lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
    print("Training accuracy of l1 logreg with C={:.3f}: {:.2f}".format(
          C, lr_l1.score(X_train, y_train)))
    print("Test accuracy of l1 logreg with C={:.3f}: {:.2f}".format(
          C, lr_l1.score(X_test, y_test)))

plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.ylim(-5, 5)
plt.legend(loc=3)
```

Out[46]:

```
Training accuracy of l1 logreg with C=0.001: 0.91
Test accuracy of l1 logreg with C=0.001: 0.92
Training accuracy of l1 logreg with C=1.000: 0.96
Test accuracy of l1 logreg with C=1.000: 0.96
Training accuracy of l1 logreg with C=100.000: 0.99
Test accuracy of l1 logreg with C=100.000: 0.98
```

如你所见，用于二分类的线性模型与用于回归的线性模型有许多相似之处。与用于回归的线性模型一样，模型的主要差别在于 `penalty` 参数，这个参数会影响正则化，也会影响模型是使用所有可用特征还是只选择特征的一个子集。

6. 用于多分类的线性模型

许多线性分类模型只适用于二分类问题，不能轻易推广到多类别问题（除了 Logistic 回归）。将二分类算法推广到多分类算法的一种常见方法是**“一对其余”**（one-vs.-rest）方法。在“一对其余”方法中，对每个类别都学习一个二分类模型，将这个类别与所有其他类别尽量分开，这样就生成了与类别个数一样多的二分类模型。在测试点上运行所有二类分类器来进行预测。在对应类别上分数最高的分类器“胜出”，将这个类别标签返回作为预测结果。

每个类别都对应一个二类分类器，这样每个类别也都有一个系数（w）向量和一个截距（b）。下面给出的是分类置信方程，其结果中最大值对应的类别即为预测的类别标签：

$$w[0] * x[0] + w[1] * x[1] + \cdots + w[p] * x[p] + b$$

多分类 Logistic 回归背后的数学与“一对其余”方法稍有不同，但它也是对每个类别都有一个系数向量和一个截距，也使用了相同的预测方法。

我们将“一对其余”方法应用在一个简单的三分类数据集上。我们用到了一个二维数据集，每个类别的数据都是从一个高斯分布中采样得出的（见图 2-19）：

In[47]:

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
```

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(["Class 0", "Class 1", "Class 2"])
```

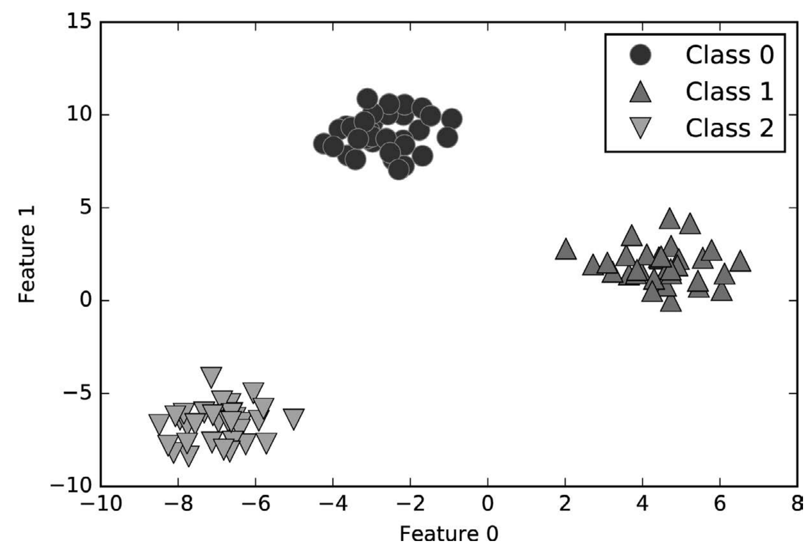


图 2-19：包含 3 个类别的二维玩具数据集

现在，在这个数据集上训练一个 LinearSVC 分类器：

In[48]:

```
linear_svm = LinearSVC().fit(X, y)
print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)
```

Out[48]:

```
Coefficient shape:  (3, 2)
Intercept shape:  (3,)
```

我们看到，coef_ 的形状是 (3, 2)，说明 coef_ 每行包含三个类别之一的系数向量，每列包含某个特征（这个数据集有 2 个特征）对应的系数值。现在 intercept_ 是一维数组，保存每个类别的截距。

我们将这 3 个二类分类器给出的直线可视化（图 2-20）：

In[49]:

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
```

```
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',
            'Line class 2'], loc=(1.01, 0.3))
```

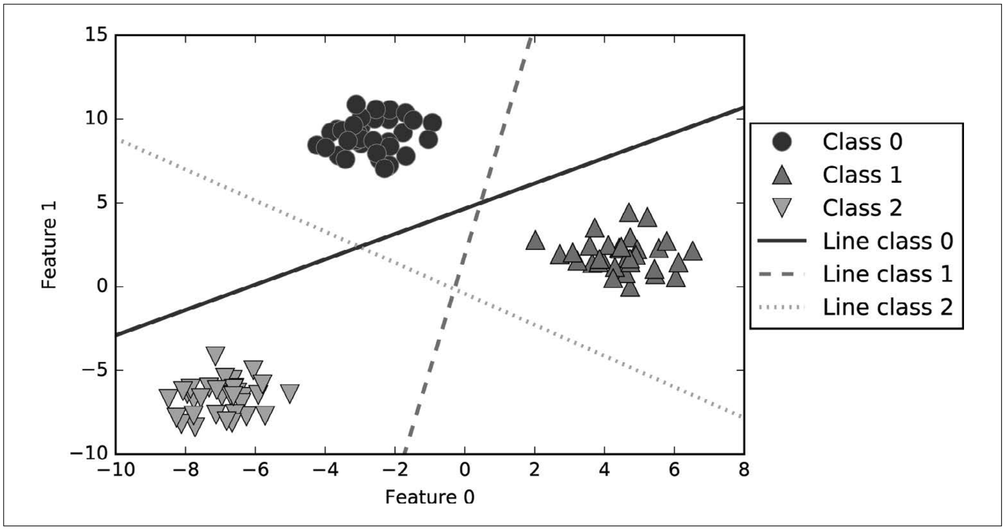


图 2-20：三个“一对其余”分类器学到的决策边界

你可以看到，训练集中所有属于类别 0 的点都在与类别 0 对应的直线上方，这说明它们位于这个二类分类器属于“类别 0”的那一侧。属于类别 0 的点位于与类别 2 对应的直线上方，这说明它们被类别 2 的二类分类器划为“其余”。属于类别 0 的点位于与类别 1 对应的直线左侧，这说明类别 1 的二元分类器将它们划为“其余”。因此，这一区域的所有点都会被最终分类器划为类别 0（类别 0 的分类器的分类置信方程的结果大于 0，其他两个类别对应的结果都小于 0）。

但图像中间的三角形区域属于哪一个类别呢，3 个二类分类器都将这一区域内的点划为“其余”。这里的点应该划归到哪一个类别呢？答案是分类方程结果最大的那个类别，即最接近的那条线对应的类别。

下面的例子（图 2-21）给出了二维空间中所有区域的预测结果：

In[50]:

```
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  ['b', 'r', 'g']):
plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',
            'Line class 2'], loc=(1.01, 0.3))
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

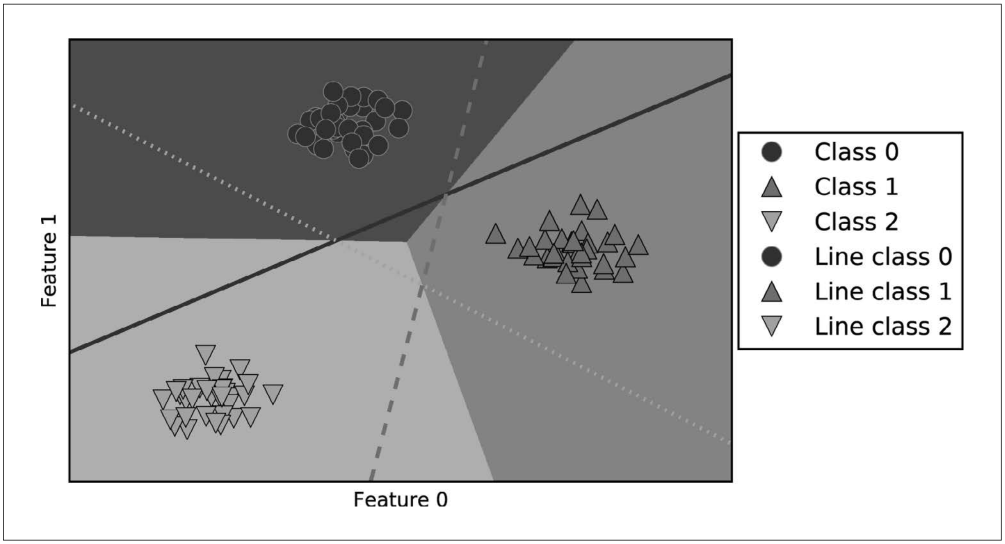


图 2-21：三个“一对其余”分类器得到的多分类决策边界

7. 优点、缺点和参数

线性模型的主要参数是正则化参数，在回归模型中叫作 `alpha`，在 `LinearSVC` 和 `Logistic-Regression` 中叫作 `C`。`alpha` 值较大或 `C` 值较小，说明模型比较简单。特别是对于回归模型而言，调节这些参数非常重要。通常在对数尺度上对 `C` 和 `alpha` 进行搜索。你还需要确定的是用 L1 正则化还是 L2 正则化。如果你假定只有几个特征是真正重要的，那么你应该用 L1 正则化，否则应默认使用 L2 正则化。如果模型的可解释性很重要的话，使用 L1 也会有帮助。由于 L1 只用到几个特征，所以更容易解释哪些特征对模型是重要的，以及这些特征的作用。

线性模型的训练速度非常快，预测速度也很快。这种模型可以推广到非常大的数据集，对稀疏数据也很有效。如果你的数据包含数十万甚至上百万个样本，你可能需要研究如何使用 `LogisticRegression` 和 `Ridge` 模型的 `solver='sag'` 选项，在处理大型数据时，这一选项比默认值要更快。其他选项还有 `SGDClassifier` 类和 `SGDRegressor` 类，它们对本节介绍的线性模型实现了可扩展性更强的版本。

线性模型的另一个优点在于，利用我们之间见过的用于回归和分类的公式，理解如何进行预测是相对比较容易的。不幸的是，往往并不完全清楚系数为什么是这样的。如果你的数据集中包含高度相关的特征，这一问题尤为突出。在这种情况下，可能很难对系数做出解释。

如果特征数量大于样本数量，线性模型的表现通常都很好。它也常用于非常大的数据集，只是因为训练其他模型并不可行。但在更低维的空间中，其他模型的泛化性能可能更好。2.3.7 节会介绍几个线性模型不适用的例子。

方法链

scikit-learn 中所有模型的 fit 方法返回的都是 self。这允许你像下面这样编写代码（我们在本章已经用过很多次了）：

In[51]:

```
# 用一行代码初始化模型并拟合
logreg = LogisticRegression().fit(X_train, y_train)
```

这里我们利用 fit 的返回值（即 self）将训练后的模型赋值给变量 logreg。这种方法调用的拼接（先调用 __init__，然后调用 fit）被称为**方法链**（method chaining）。scikit-learn 中方法链的另一个常见用法是在一行代码中同时 fit 和 predict：

In[52]:

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

最后，你甚至可以在一行代码中完成模型初始化、拟合和预测：

In[53]:

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

不过这种非常简短的写法并不完美。一行代码中发生了很多事情，可能会使代码变得难以阅读。此外，拟合后的回归模型也没有保存在任何变量中，所以我们既不能查看它也不能用它来预测其他数据。

2.3.4　朴素贝叶斯分类器

朴素贝叶斯分类器是与上一节介绍的线性模型非常相似的一种分类器，但它的训练速度往往更快。这种高效率所付出的代价是，朴素贝叶斯模型的泛化能力要比线性分类器（如 LogisticRegression 和 LinearSVC）稍差。

朴素贝叶斯模型如此高效的原因在于，它通过单独查看每个特征来学习参数，并从每个特征中收集简单的类别统计数据。scikit-learn 中实现了三种朴素贝叶斯分类器：GaussianNB、BernoulliNB 和 MultinomialNB。GaussianNB 可应用于任意连续数据，而 BernoulliNB 假定输入数据为二分类数据，MultinomialNB 假定输入数据为计数数据（即每个特征代表某个对象的整数计数，比如一个单词在句子里出现的次数）。BernoulliNB 和 MultinomialNB 主要用于文本数据分类。

BernoulliNB 分类器计算每个类别中每个特征不为 0 的元素个数。用一个例子来说明会很容易理解：

In[54]:

```
X = np.array([[0, 1, 0, 1],
              [1, 0, 1, 1],
              [0, 0, 0, 1],
              [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

这里我们有 4 个数据点，每个点有 4 个二分类特征。一共有两个类别：0 和 1。对于类别 0（第 1、3 个数据点），第一个特征有 2 个为零、0 个不为零，第二个特征有 1 个为零、1 个不为零，以此类推。然后对类别 1 中的数据点计算相同的计数。计算每个类别中的非零元素个数，大体上看起来像这样：

In[55]:

```
counts = {}
for label in np.unique(y):
    # 对每个类别进行遍历
    # 计算（求和）每个特征中1的个数
    counts[label] = X[y == label].sum(axis=0)
print("Feature counts:\n{}".format(counts))
```

Out[55]:

```
Feature counts:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

另外两种朴素贝叶斯模型（MultinomialNB 和 GaussianNB）计算的统计数据类型略有不同。MultinomialNB 计算每个类别中每个特征的平均值，而 GaussianNB 会保存每个类别中每个特征的平均值和标准差。

要想做出预测，需要将数据点与每个类别的统计数据进行比较，并将最匹配的类别作为预测结果。有趣的是，MultinomialNB 和 BernoulliNB 预测公式的形式都与线性模型完全相同（见 2.3.3 节）。不幸的是，朴素贝叶斯模型 coef_ 的含义与线性模型稍有不同，因为 coef_ 不同于 w。

优点、缺点和参数

MultinomialNB 和 BernoulliNB 都只有一个参数 alpha，用于控制模型复杂度。alpha 的工作原理是，算法向数据中添加 alpha 这么多的虚拟数据点，这些点对所有特征都取正值。这可以将统计数据“平滑化”（smoothing）。alpha 越大，平滑化越强，模型复杂度就越低。算法性能对 alpha 值的鲁棒性相对较好，也就是说，alpha 值对模型性能并不重要。但调整这个参数通常都会使精度略有提高。

GaussianNB 主要用于高维数据，而另外两种朴素贝叶斯模型则广泛用于稀疏计数数据，比如文本。MultinomialNB 的性能通常要优于 BernoulliNB，特别是在包含很多非零特征的数据集（即大型文档）上。

朴素贝叶斯模型的许多优点和缺点都与线性模型相同。它的训练和预测速度都很快，训练过程也很容易理解。该模型对高维稀疏数据的效果很好，对参数的鲁棒性也相对较好。朴素贝叶斯模型是很好的基准模型，常用于非常大的数据集，在这些数据集上即使训练线性模型可能也要花费大量时间。

2.3.5　决策树

决策树是广泛用于分类和回归任务的模型。本质上，它从一层层的 if/else 问题中进行学习，并得出结论。

这些问题类似于你在“20 Questions”游戏[9]中可能会问的问题。想象一下，你想要区分下面这四种动物：熊、鹰、企鹅和海豚。你的目标是通过提出尽可能少的 if/else 问题来得到正确答案。你可能首先会问：这种动物有没有羽毛，这个问题会将可能的动物减少到只有两种。如果答案是“有”，你可以问下一个问题，帮你区分鹰和企鹅。例如，你可以问这种动物会不会飞。如果这种动物没有羽毛，那么可能是海豚或熊，所以你需要问一个问题来区分这两种动物——比如问这种动物有没有鳍。

这一系列问题可以表示为一棵决策树，如图 2-22 所示。

In[56]:

```
mglearn.plots.plot_animal_tree()
```

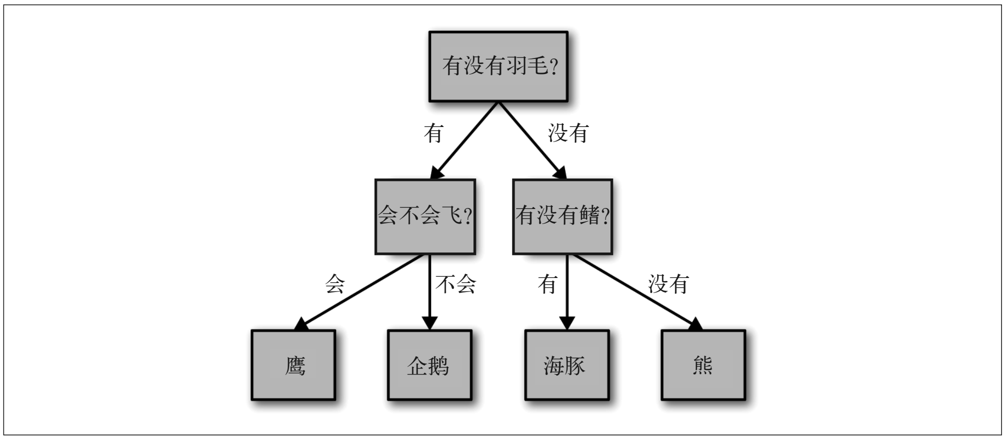


图 2-22：区分几种动物的决策树

在这张图中，树的每个结点代表一个问题或一个包含答案的终结点（也叫**叶结点**）。树的边将问题的答案与将问的下一个问题连接起来。

用机器学习的语言来说就是，为了区分四类动物（鹰、企鹅、海豚和熊），我们利用三个特征（“有没有羽毛”“会不会飞”和“有没有鳍”）来构建一个模型。我们可以利用监督学习从数据中学习模型，而无需人为构建模型。

1. 构造决策树

我们在图 2-23 所示的二维分类数据集上构造决策树。这个数据集由 2 个半月形组成，每个类别都包含 50 个数据点。我们将这个数据集称为 `two_moons`。

学习决策树，就是学习一系列 if/else 问题，使我们能够以最快的速度得到正确答案。在机器学习中，这些问题叫作**测试**（不要与测试集弄混，测试集是用来测试模型泛化性能的数据）。数据通常并不是像动物的例子那样具有二元特征（是 / 否）的形式，而是表示为连续特征，比如图 2-23 所示的二维数据集。用于连续数据的测试形式是：“特征 i 的值是否大于 a？”

注 9：一种室内游戏，其中一人想象一个对象，其他人轮流通过向他提问来猜测这个对象，他只能回答“是”或“否”。如果 20 轮问题过后仍没人猜出，则这个人获胜。——译者注

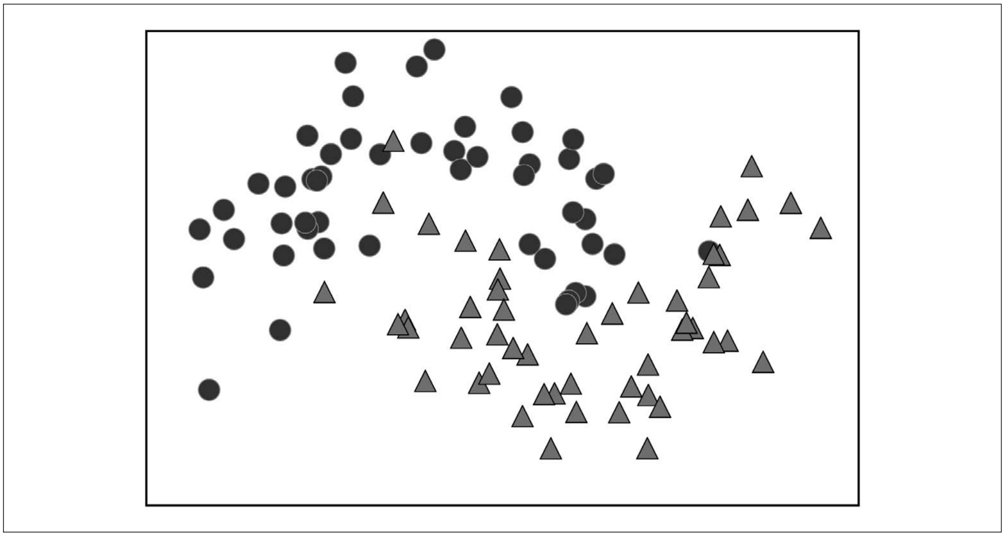


图 2-23：用于构造决策树的 two_moons 数据集

为了构造决策树，算法搜遍所有可能的测试，找出对目标变量来说信息量最大的那一个。图 2-24 展示了选出的第一个测试。将数据集在 x[1]=0.0596 处垂直划分可以得到最多信息，它在最大程度上将类别 0 中的点与类别 1 中的点进行区分。顶结点（也叫**根结点**）表示整个数据集，包含属于类别 0 的 50 个点和属于类别 1 的 50 个点。通过测试 x[1] <= 0.0596 的真假来对数据集进行划分，在图中表示为一条黑线。如果测试结果为真，那么将这个点分配给左结点，左结点里包含属于类别 0 的 2 个点和属于类别 1 的 32 个点。否则将这个点分配给右结点，右结点里包含属于类别 0 的 48 个点和属于类别 1 的 18 个点。这两个结点对应于图 2-24 中的顶部区域和底部区域。尽管第一次划分已经对两个类别做了很好的区分，但底部区域仍包含属于类别 0 的点，顶部区域也仍包含属于类别 1 的点。我们可以在两个区域中重复寻找最佳测试的过程，从而构建出更准确的模型。图 2-25 展示了信息量最大的下一次划分，这次划分是基于 x[0] 做出的，分为左右两个区域。

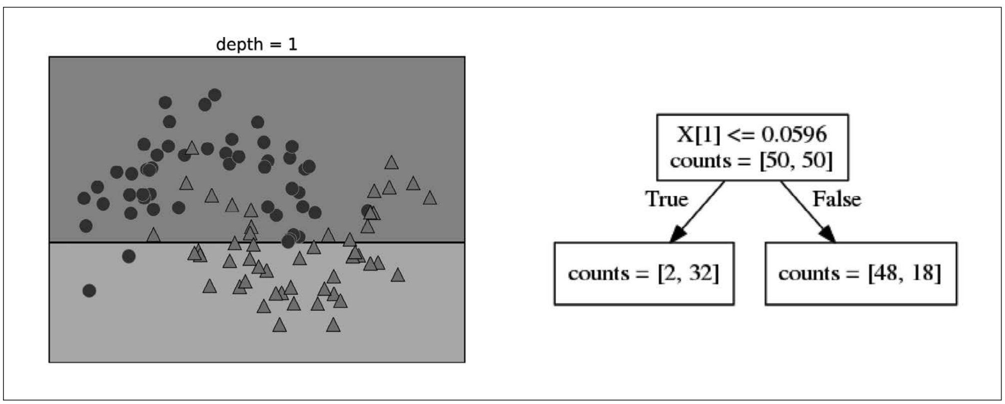


图 2-24：深度为 1 的树的决策边界（左）与相应的树（右）

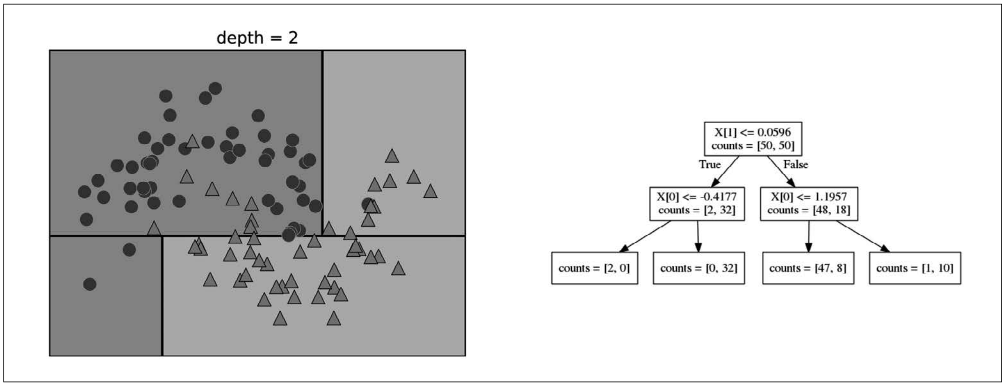


图 2-25：深度为 2 的树的决策边界（左）与相应的树（右）

这一递归过程生成一棵二元决策树，其中每个结点都包含一个测试。或者你可以将每个测试看成沿着一条轴对当前数据进行划分。这是一种将算法看作分层划分的观点。由于每个测试仅关注一个特征，所以划分后的区域边界始终与坐标轴平行。

对数据反复进行递归划分，直到划分后的每个区域（决策树的每个叶结点）只包含单一目标值（单一类别或单一回归值）。如果树中某个叶结点所包含数据点的目标值都相同，那么这个叶结点就是**纯的**（pure）。这个数据集的最终划分结果见图 2-26。

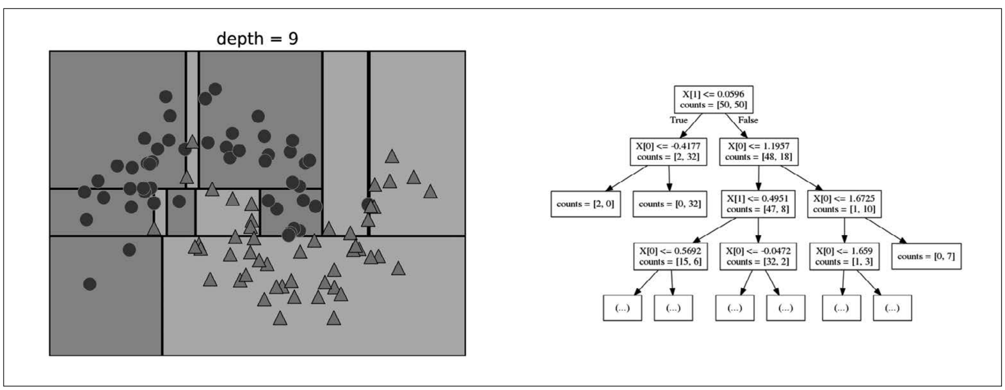


图 2-26：深度为 9 的树的决策边界（左）与相应的树的一部分（右）；完整的决策树非常大，很难可视化

想要对新数据点进行预测，首先要查看这个点位于特征空间划分的哪个区域，然后将该区域的多数目标值（如果是纯的叶结点，就是单一目标值）作为预测结果。从根结点开始对树进行遍历就可以找到这一区域，每一步向左还是向右取决于是否满足相应的测试。

决策树也可以用于回归任务，使用的方法完全相同。预测的方法是，基于每个结点的测试对树进行遍历，最终找到新数据点所属的叶结点。这一数据点的输出即为此叶结点中所有训练点的平均目标值。

2. 控制决策树的复杂度

通常来说，构造决策树直到所有叶结点都是纯的叶结点，这会导致模型非常复杂，并且对训练数据高度过拟合。纯叶结点的存在说明这棵树在训练集上的精度是 100%。训练集中的每个数据点都位于分类正确的叶结点中。在图 2-26 的左图中可以看出过拟合。你可以看到，在所有属于类别 0 的点中间有一块属于类别 1 的区域。另一方面，有一小条属于类别 0 的区域，包围着最右侧属于类别 0 的那个点。这并不是人们想象中决策边界的样子，这个决策边界过于关注远离同类别其他点的单个异常点。

防止过拟合有两种常见的策略：一种是及早停止树的生长，也叫**预剪枝**（pre-pruning）；另一种是先构造树，但随后删除或折叠信息量很少的结点，也叫**后剪枝**（post-pruning）或**剪枝**（pruning）。预剪枝的限制条件可能包括限制树的最大深度、限制叶结点的最大数目，或者规定一个结点中数据点的最小数目来防止继续划分。

`scikit-learn` 的决策树在 `DecisionTreeRegressor` 类和 `DecisionTreeClassifier` 类中实现。`scikit-learn` 只实现了预剪枝，没有实现后剪枝。

我们在乳腺癌数据集上更详细地看一下预剪枝的效果。和前面一样，我们导入数据集并将其分为训练集和测试集。然后利用默认设置来构建模型，默认将树完全展开（树不断分支，直到所有叶结点都是纯的）。我们固定树的 `random_state`，用于在内部解决平局问题：

In[58]:

```
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Out[58]:

```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

不出所料，训练集上的精度是 100%，这是因为叶结点都是纯的，树的深度很大，足以完美地记住训练数据的所有标签。测试集精度比之前讲过的线性模型略低，线性模型的精度约为 95%。

如果我们不限制决策树的深度，它的深度和复杂度都可以变得特别大。因此，未剪枝的树容易过拟合，对新数据的泛化性能不佳。现在我们将预剪枝应用在决策树上，这可以在完美拟合训练数据之前阻止树的展开。一种选择是在到达一定深度后停止树的展开。这里我们设置 `max_depth=4`，这意味着只可以连续问 4 个问题（参见图 2-24 和图 2-26）。限制树的深度可以减少过拟合。这会降低训练集的精度，但可以提高测试集的精度：

In[59]:

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)
```

```
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Out[59]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

3. 分析决策树

我们可以利用 `tree` 模块的 `export_graphviz` 函数来将树可视化。这个函数会生成一个 .dot 格式的文件，这是一种用于保存图形的文本文件格式。我们设置为结点添加颜色的选项，颜色表示每个结点中的多数类别，同时传入类别名称和特征名称，这样可以对树正确标记：

In[60]:

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant","benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

我们可以利用 `graphviz` 模块读取这个文件并将其可视化（你也可以使用任何能够读取 .dot 文件的程序），见图 2-27：

In[61]:

```
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

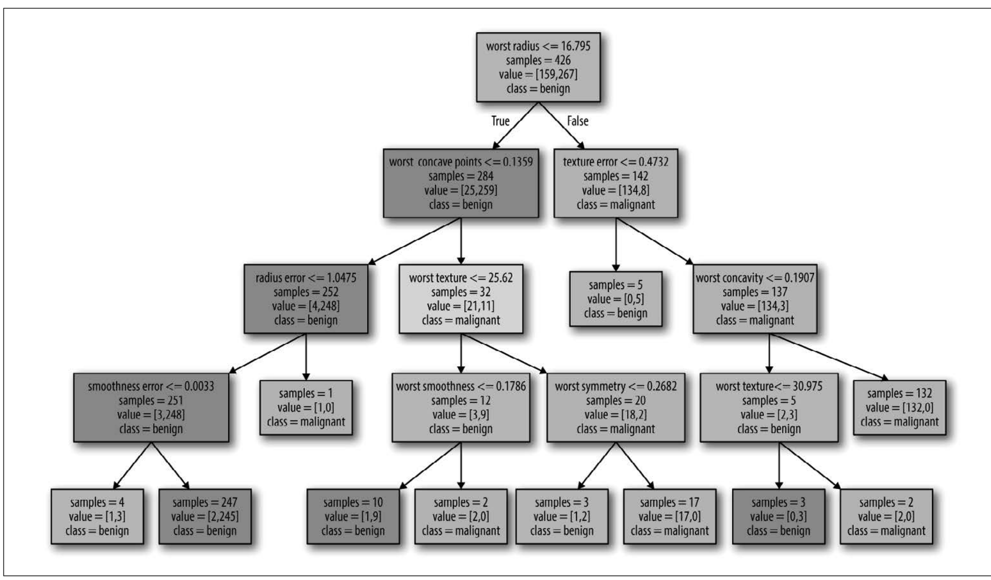


图 2-27：基于乳腺癌数据集构造的决策树的可视化

树的可视化有助于深入理解算法是如何进行预测的，也是易于向非专家解释的机器学习算法的优秀示例。不过，即使这里树的深度只有 4 层，也有点太大了。深度更大的树（深度为 10 并不罕见）更加难以理解。一种观察树的方法可能有用，就是找出大部分数据的实际路径。图 2-27 中每个结点的 `samples` 给出了该结点中的样本个数，`values` 给出的是每个类别的样本个数。观察 `worst radius <= 16.795` 分支右侧的子结点，我们发现它只包含 8 个良性样本，但有 134 个恶性样本。树的这一侧的其余分支只是利用一些更精细的区别将这 8 个良性样本分离出来。在第一次划分右侧的 142 个样本中，几乎所有样本（132 个）最后都进入最右侧的叶结点中。

再来看一下根结点的左侧子结点，对于 `worst radius > 16.795`，我们得到 25 个恶性样本和 259 个良性样本。几乎所有良性样本最终都进入左数第二个叶结点中，大部分其他叶结点都只包含很少的样本。

4. 树的特征重要性

查看整个树可能非常费劲，除此之外，我还可以利用一些有用的属性来总结树的工作原理。其中最常用的是**特征重要性**（feature importance），它为每个特征对树的决策的重要性进行排序。对于每个特征来说，它都是一个介于 0 和 1 之间的数字，其中 0 表示“根本没用到”，1 表示“完美预测目标值”。特征重要性的求和始终为 1：

In[62]:

```
print("Feature importances:\n{}".format(tree.feature_importances_))
```

Out[62]:

```
Feature importances:
[ 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.     0.01
  0.048  0.     0.     0.002  0.     0.     0.     0.     0.     0.727  0.046
  0.     0.     0.014  0.     0.018  0.122  0.012  0.   ]
```

我们可以将特征重要性可视化，与我们将线性模型的系数可视化的方法类似（图 2-28）：

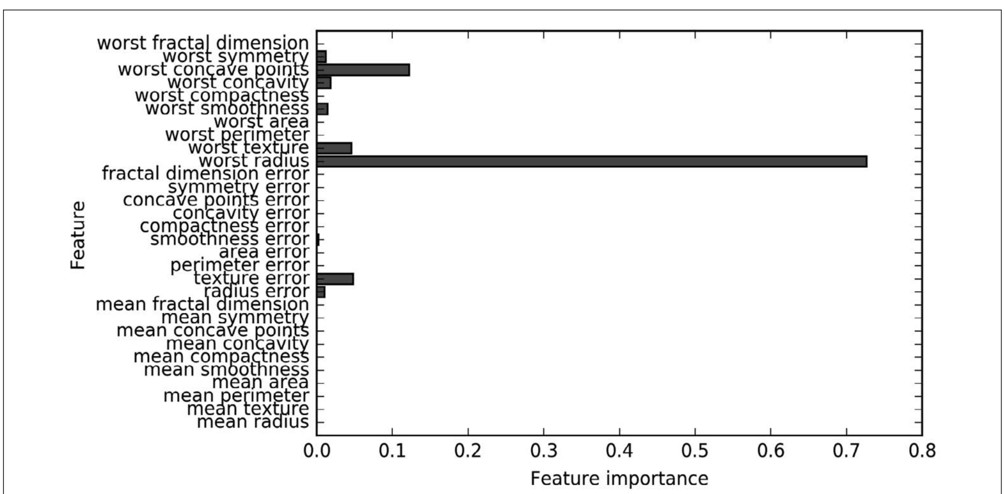


图 2-28：在乳腺癌数据集上学到的决策树的特征重要性

In[63]:

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")

plot_feature_importances_cancer(tree)
```

这里我们看到，顶部划分用到的特征（“worst radius”）是最重要的特征。这也证实了我们在分析树时的观察结论，即第一层划分已经将两个类别区分得很好。

但是，如果某个特征的 feature_importance_ 很小，并不能说明这个特征没有提供任何信息。这只能说明该特征没有被树选中，可能是因为另一个特征也包含了同样的信息。

与线性模型的系数不同，特征重要性始终为正数，也不能说明该特征对应哪个类别。特征重要性告诉我们“worst radius”（最大半径）特征很重要，但并没有告诉我们半径大表示样本是良性还是恶性。事实上，在特征和类别之间可能没有这样简单的关系，你可以在下面的例子中看出这一点（图 2-29 和图 2-30）：

In[64]:

```
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)
```

Out[64]:

```
Feature importances: [ 0.  1.]
```

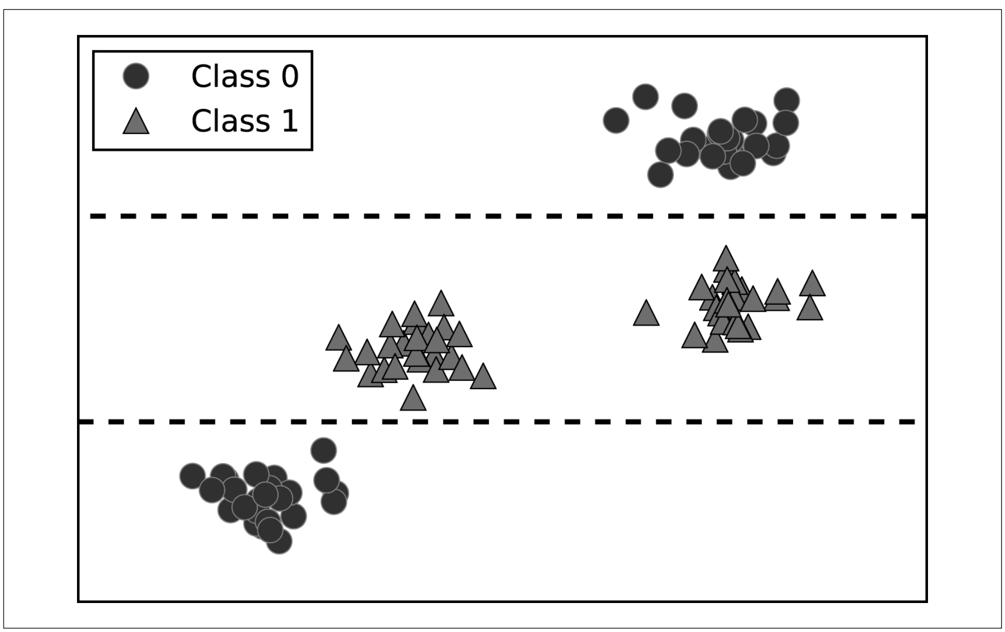


图 2-29：一个二维数据集（*y* 轴上的特征与类别标签是非单调的关系）与决策树给出的决策边界

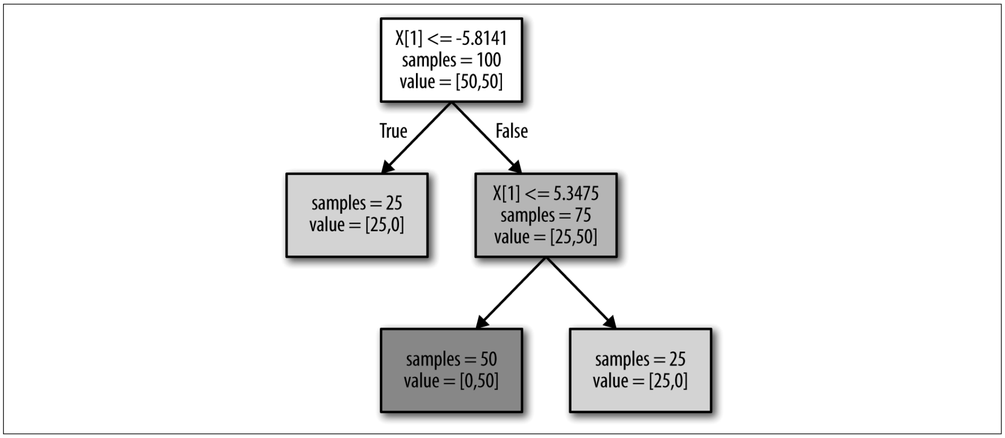


图 2-30：从图 2-29 的数据中学到的决策树

该图显示的是有两个特征和两个类别的数据集。这里所有信息都包含在 `X[1]` 中，没有用到 `X[0]`。但 `X[1]` 和输出类别之间并不是单调关系，即我们不能这么说："较大的 `X[1]` 对应类别 0，较小的 `X[1]` 对应类别 1"（反之亦然）。

虽然我们主要讨论的是用于分类的决策树，但对用于回归的决策树来说，所有内容都是类似的，在 `DecisionTreeRegressor` 中实现。回归树的用法和分析与分类树非常类似。但在将基于树的模型用于回归时，我们想要指出它的一个特殊性质。`DecisionTreeRegressor`（以及其他所有基于树的回归模型）不能**外推**（extrapolate），也不能在训练数据范围之外进行预测。

我们利用计算机内存（RAM）历史价格的数据集来更详细地研究这一点。图 2-31 给出了这个数据集的图像，x 轴为日期，y 轴为那一年 1 兆字节（MB）RAM 的价格：

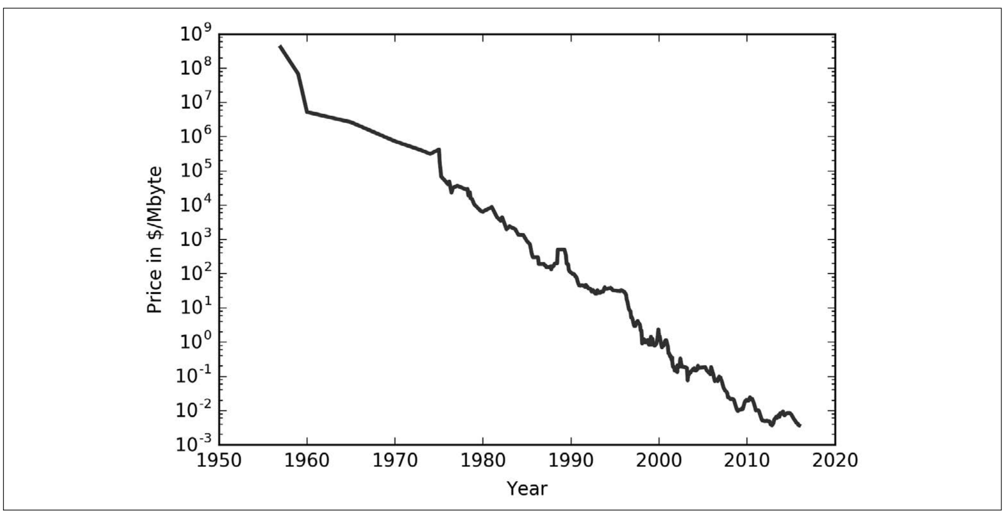


图 2-31：用对数坐标绘制 RAM 价格的历史发展

In[65]:

```
import pandas as pd
ram_prices = pd.read_csv("data/ram_price.csv")

plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Year")
plt.ylabel("Price in $/Mbyte")
```

注意 y 轴的对数刻度。在用对数坐标绘图时，二者的线性关系看起来非常好，所以预测应该相对比较容易，除了一些不平滑之处之外。

我们将利用 2000 年前的历史数据来预测 2000 年后的价格，只用日期作为特征。我们将对比两个简单的模型：`DecisionTreeRegressor` 和 `LinearRegression`。我们对价格取对数，使得二者关系的线性相对更好。这对 `DecisionTreeRegressor` 不会产生什么影响，但对 `LinearRegression` 的影响却很大（我们将在第 4 章中进一步讨论）。训练模型并做出预测之后，我们应用指数映射来做对数变换的逆运算。为了便于可视化，我们这里对整个数据集进行预测，但如果是为了定量评估，我们将只考虑测试数据集：

In[66]:

```
from sklearn.tree import DecisionTreeRegressor
# 利用历史数据预测2000年后的价格
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]

# 基于日期来预测价格
X_train = data_train.date[:, np.newaxis]
# 我们利用对数变换得到数据和目标之间更简单的关系
y_train = np.log(data_train.price)

tree = DecisionTreeRegressor().fit(X_train, y_train)
linear_reg = LinearRegression().fit(X_train, y_train)

# 对所有数据进行预测
X_all = ram_prices.date[:, np.newaxis]

pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)

# 对数变换逆运算
price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

这里创建的图 2-32 将决策树和线性回归模型的预测结果与真实值进行对比：

In[67]:

```
plt.semilogy(data_train.date, data_train.price, label="Training data")
plt.semilogy(data_test.date, data_test.price, label="Test data")
plt.semilogy(ram_prices.date, price_tree, label="Tree prediction")
plt.semilogy(ram_prices.date, price_lr, label="Linear prediction")
plt.legend()
```

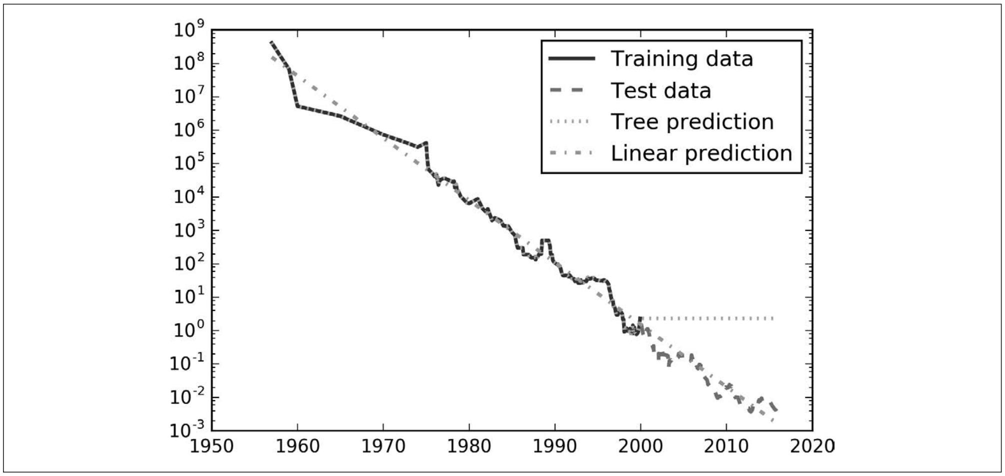


图 2-32：线性模型和回归树对 RAM 价格数据的预测结果对比

两个模型之间的差异非常明显。线性模型用一条直线对数据做近似，这是我们所知道的。这条线对测试数据（2000 年后的价格）给出了相当好的预测，不过忽略了训练数据和测试数据中一些更细微的变化。与之相反，树模型完美预测了训练数据。由于我们没有限制树的复杂度，因此它记住了整个数据集。但是，一旦输入超出了模型训练数据的范围，模型就只能持续预测最后一个已知数据点。树不能在训练数据的范围之外生成"新的"响应。所有基于树的模型都有这个缺点。[10]

5. 优点、缺点和参数

如前所述，控制决策树模型复杂度的参数是预剪枝参数，它在树完全展开之前停止树的构造。通常来说，选择一种预剪枝策略（设置 `max_depth`、`max_leaf_nodes` 或 `min_samples_leaf`）足以防止过拟合。

与前面讨论过的许多算法相比，决策树有两个优点：一是得到的模型很容易可视化，非专家也很容易理解（至少对于较小的树而言）；二是算法完全不受数据缩放的影响。由于每个特征被单独处理，而且数据的划分也不依赖于缩放，因此决策树算法不需要特征预处理，比如归一化或标准化。特别是特征的尺度完全不一样时或者二元特征和连续特征同时存在时，决策树的效果很好。

决策树的主要缺点在于，即使做了预剪枝，它也经常会过拟合，泛化性能很差。因此，在大多数应用中，往往使用下面介绍的集成方法来替代单棵决策树。

2.3.6　决策树集成

集成（ensemble）是合并多个机器学习模型来构建更强大模型的方法。在机器学习文献中有许多模型都属于这一类，但已证明有两种集成模型对大量分类和回归的数据集都是

注 10：实际上，利用基于树的模型可以做出非常好的预测（比如试图预测价格会上涨还是下跌）。这个例子的目的并不是要说明对时间序列来说树是一个不好的模型，而是为了说明树在预测方式上的特殊性质。

有效的，二者都以决策树为基础，分别是随机森林（random forest）和梯度提升决策树（gradient boosted decision tree）。

1. 随机森林

我们刚刚说过，决策树的一个主要缺点在于经常对训练数据过拟合。随机森林是解决这个问题的一种方法。随机森林本质上是许多决策树的集合，其中每棵树都和其他树略有不同。随机森林背后的思想是，每棵树的预测可能都相对较好，但可能对部分数据过拟合。如果构造很多树，并且每棵树的预测都很好，但都以不同的方式过拟合，那么我们可以对这些树的结果取平均值来降低过拟合。既能减少过拟合又能保持树的预测能力，这可以在数学上严格证明。

为了实现这一策略，我们需要构造许多决策树。每棵树都应该对目标值做出可以接受的预测，还应该与其他树不同。随机森林的名字来自于将随机性添加到树的构造过程中，以确保每棵树都各不相同。随机森林中树的随机化方法有两种：一种是通过选择用于构造树的数据点，另一种是通过选择每次划分测试的特征。我们来更深入地研究这一过程。

构造随机森林。想要构造一个随机森林模型，你需要确定用于构造的树的个数（`RandomForestRegressor` 或 `RandomForestClassifier` 的 `n_estimators` 参数）。比如我们想要构造 10 棵树。这些树在构造时彼此完全独立，算法对每棵树进行不同的随机选择，以确保树和树之间是有区别的。想要构造一棵树，首先要对数据进行**自助采样**（bootstrap sample）。也就是说，从 `n_samples` 个数据点中有放回地（即同一样本可以被多次抽取）重复随机抽取一个样本，共抽取 `n_samples` 次。这样会创建一个与原数据集大小相同的数据集，但有些数据点会缺失（大约三分之一），有些会重复。

举例说明，比如我们想要创建列表 `['a', 'b', 'c', 'd']` 的自助采样。一种可能的自主采样是 `['b', 'd', 'd', 'c']`，另一种可能的采样为 `['d', 'a', 'd', 'a']`。

接下来，基于这个新创建的数据集来构造决策树。但是，要对我们在介绍决策树时描述的算法稍作修改。在每个结点处，算法随机选择特征的一个子集，并对其中一个特征寻找最佳测试，而不是对每个结点都寻找最佳测试。选择的特征个数由 `max_features` 参数来控制。每个结点中特征子集的选择是相互独立的，这样树的每个结点可以使用特征的不同子集来做出决策。

由于使用了自助采样，随机森林中构造每棵决策树的数据集都是略有不同的。由于每个结点的特征选择，每棵树中的每次划分都是基于特征的不同子集。这两种方法共同保证随机森林中所有树都不相同。

在这个过程中的一个关键参数是 `max_features`。如果我们设置 `max_features` 等于 `n_features`，那么每次划分都要考虑数据集的所有特征，在特征选择的过程中没有添加随机性（不过自助采样依然存在随机性）。如果设置 `max_features` 等于 `1`，那么在划分时将无法选择对哪个特征进行测试，只能对随机选择的某个特征搜索不同的阈值。因此，如果 `max_features` 较大，那么随机森林中的树将会十分相似，利用最独特的特征可以轻松拟合数据。如果 `max_features` 较小，那么随机森林中的树将会差异很大，为了很好地拟合数据，每棵树的深度都要很大。

想要利用随机森林进行预测，算法首先对森林中的每棵树进行预测。对于回归问题，我们可以对这些结果取平均值作为最终预测。对于分类问题，则用到了“软投票”（soft voting）策略。也就是说，每个算法做出“软”预测，给出每个可能的输出标签的概率。对所有树的预测概率取平均值，然后将概率最大的类别作为预测结果。

分析随机森林。下面将由 5 棵树组成的随机森林应用到前面研究过的 two_moons 数据集上：

In[68]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

作为随机森林的一部分，树被保存在 estimator_ 属性中。我们将每棵树学到的决策边界可视化，也将它们的总预测（即整个森林做出的预测）可视化（图 2-33）：

In[69]:

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
                                alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

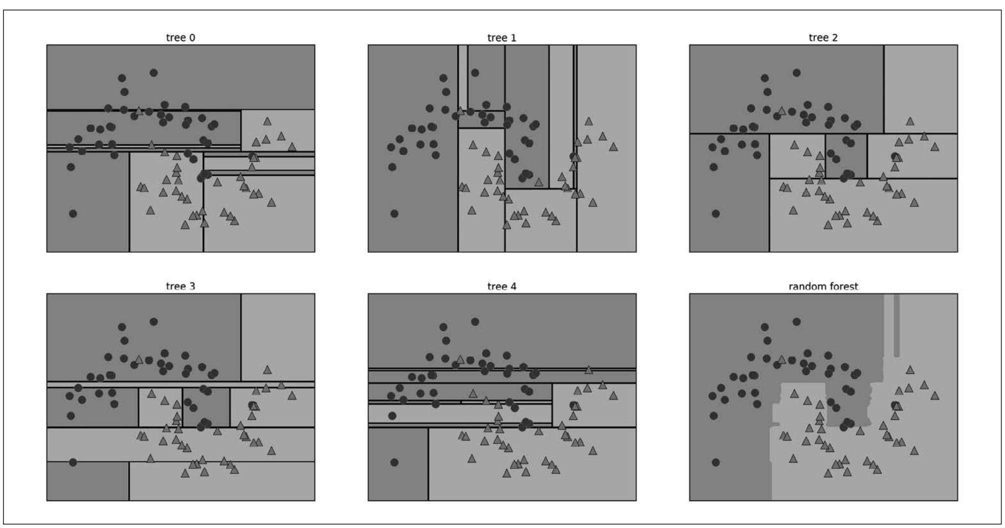


图 2-33：5 棵随机化的决策树找到的决策边界，以及将它们的预测概率取平均后得到的决策边界

你可以清楚地看到，这 5 棵树学到的决策边界大不相同。每棵树都犯了一些错误，因为这里画出的一些训练点实际上并没有包含在这些树的训练集中，原因在于自助采样。

随机森林比单独每一棵树的过拟合都要小，给出的决策边界也更符合直觉。在任何实际应用中，我们会用到更多棵树（通常是几百或上千），从而得到更平滑的边界。

再举一个例子，我们将包含 100 棵树的随机森林应用在乳腺癌数据集上：

In[70]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

Out[70]:

```
Accuracy on training set: 1.000
Accuracy on test set: 0.972
```

在没有调节任何参数的情况下，随机森林的精度为 97%，比线性模型或单棵决策树都要好。我们可以调节 `max_features` 参数，或者像单棵决策树那样进行预剪枝。但是，随机森林的默认参数通常就已经可以给出很好的结果。

与决策树类似，随机森林也可以给出特征重要性，计算方法是将森林中所有树的特征重要性求和并取平均。一般来说，随机森林给出的特征重要性要比单棵树给出的更为可靠。参见图 2-34。

In[71]:

```
plot_feature_importances_cancer(forest)
```

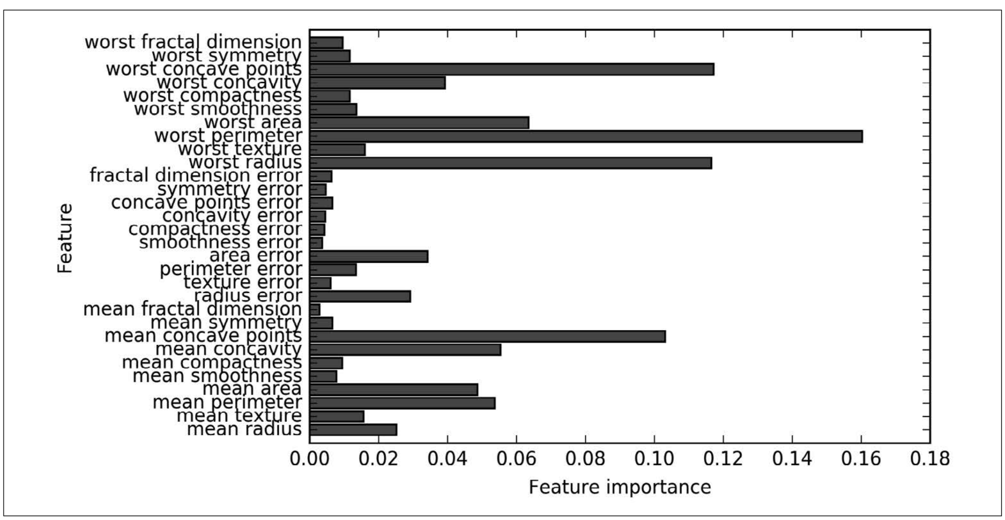


图 2-34：拟合乳腺癌数据集得到的随机森林的特征重要性

如你所见，与单棵树相比，随机森林中有更多特征的重要性不为零。与单棵决策树类似，随机森林也给了“worst radius”（最大半径）特征很大的重要性，但从总体来看，它实际上却选择“worst perimeter”（最大周长）作为信息量最大的特征。由于构造随机森林过程中的随机性，算法需要考虑多种可能的解释，结果就是随机森林比单棵树更能从总体把握数据的特征。

优点、缺点和参数。用于回归和分类的随机森林是目前应用最广泛的机器学习方法之一。这种方法非常强大，通常不需要反复调节参数就可以给出很好的结果，也不需要对数据进行缩放。

从本质上看，随机森林拥有决策树的所有优点，同时弥补了决策树的一些缺陷。仍然使用决策树的一个原因是需要决策过程的紧凑表示。基本上不可能对几十棵甚至上百棵树做出详细解释，随机森林中树的深度往往比决策树还要大（因为用到了特征子集）。因此，如果你需要以可视化的方式向非专家总结预测过程，那么选择单棵决策树可能更好。虽然在大型数据集上构建随机森林可能比较费时间，但在一台计算机的多个 CPU 内核上并行计算也很容易。如果你用的是多核处理器（几乎所有的现代化计算机都是），你可以用 `n_jobs` 参数来调节使用的内核个数。使用更多的 CPU 内核，可以让速度线性增加（使用 2 个内核，随机森林的训练速度会加倍），但设置 `n_jobs` 大于内核个数是没有用的。你可以设置 `n_jobs=-1` 来使用计算机的所有内核。

你应该记住，随机森林本质上是随机的，设置不同的随机状态（或者不设置 `random_state` 参数）可以彻底改变构建的模型。森林中的树越多，它对随机状态选择的鲁棒性就越好。如果你希望结果可以重现，固定 `random_state` 是很重要的。

对于维度非常高的稀疏数据（比如文本数据），随机森林的表现往往不是很好。对于这种数据，使用线性模型可能更合适。即使是非常大的数据集，随机森林的表现通常也很好，训练过程很容易并行在功能强大的计算机的多个 CPU 内核上。不过，随机森林需要更大的内存，训练和预测的速度也比线性模型要慢。对一个应用来说，如果时间和内存很重要的话，那么换用线性模型可能更为明智。

需要调节的重要参数有 `n_estimators` 和 `max_features`，可能还包括预剪枝选项（如 `max_depth`）。`n_estimators` 总是越大越好。对更多的树取平均可以降低过拟合，从而得到鲁棒性更好的集成。不过收益是递减的，而且树越多需要的内存也越多，训练时间也越长。常用的经验法则就是“在你的时间 / 内存允许的情况下尽量多”。

前面说过，`max_features` 决定每棵树的随机性大小，较小的 `max_features` 可以降低过拟合。一般来说，好的经验就是使用默认值：对于分类，默认值是 `max_features=sqrt(n_features)`；对于回归，默认值是 `max_features=n_features`。增大 `max_features` 或 `max_leaf_nodes` 有时也可以提高性能。它还可以大大降低用于训练和预测的时间和空间要求。

2. 梯度提升回归树（梯度提升机）

梯度提升回归树是另一种集成方法，通过合并多个决策树来构建一个更为强大的模型。虽然名字中含有“回归”，但这个模型既可以用于回归也可以用于分类。与随机森林方法不同，梯度提升采用连续的方式构造树，每棵树都试图纠正前一棵树的错误。默认情况下，梯度提升回归树中没有随机化，而是用到了强预剪枝。梯度提升树通常使用深度很小（1

到 5 之间）的树，这样模型占用的内存更少，预测速度也更快。

梯度提升背后的主要思想是合并许多简单的模型（在这个语境中叫作**弱学习器**），比如深度较小的树。每棵树只能对部分数据做出好的预测，因此，添加的树越来越多，可以不断迭代提高性能。

梯度提升树经常是机器学习竞赛的优胜者，并且广泛应用于业界。与随机森林相比，它通常对参数设置更为敏感，但如果参数设置正确的话，模型精度更高。

除了预剪枝与集成中树的数量之外，梯度提升的另一个重要参数是 `learning_rate`（学习率），用于控制每棵树纠正前一棵树的错误的强度。较高的学习率意味着每棵树都可以做出较强的修正，这样模型更为复杂。通过增大 `n_estimators` 来向集成中添加更多树，也可以增加模型复杂度，因为模型有更多机会纠正训练集上的错误。

下面是在乳腺癌数据集上应用 `GradientBoostingClassifier` 的示例。默认使用 100 棵树，最大深度是 3，学习率为 0.1：

In[72]:

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Out[72]:

```
Accuracy on training set: 1.000
Accuracy on test set: 0.958
```

由于训练集精度达到 100%，所以很可能存在过拟合。为了降低过拟合，我们可以限制最大深度来加强预剪枝，也可以降低学习率：

In[73]:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Out[73]:

```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

In[74]:

```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
```

```
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Out[74]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

降低模型复杂度的两种方法都降低了训练集精度，这和预期相同。在这个例子中，减小树的最大深度显著提升了模型性能，而降低学习率仅稍稍提高了泛化性能。

对于其他基于决策树的模型，我们也可以将特征重要性可视化，以便更好地理解模型（图 2-35）。由于我们用到了 100 棵树，所以即使所有树的深度都是 1，查看所有树也是不现实的：

In[75]:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

plot_feature_importances_cancer(gbrt)
```

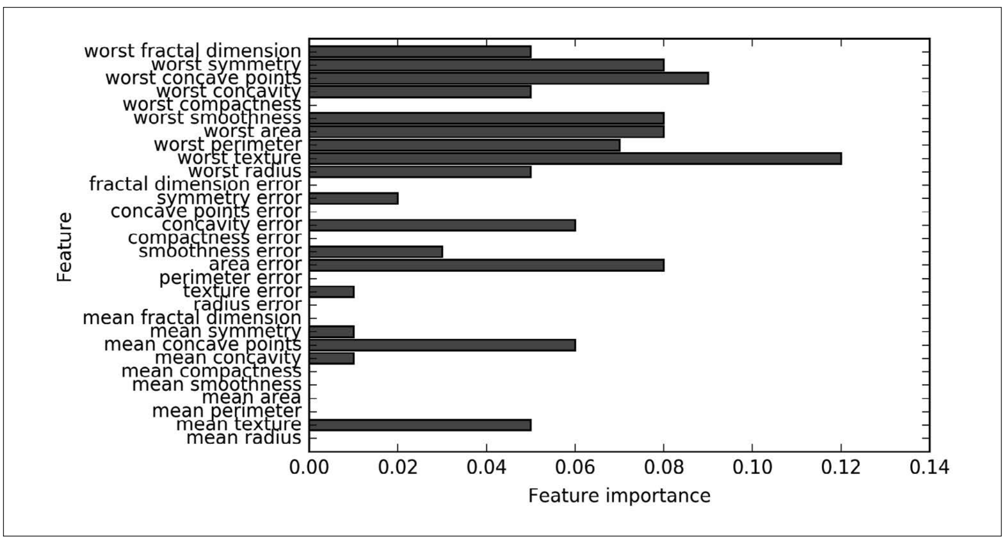


图 2-35：用于拟合乳腺癌数据集的梯度提升分类器给出的特征重要性

可以看到，梯度提升树的特征重要性与随机森林的特征重要性有些类似，不过梯度提升完全忽略了某些特征。

由于梯度提升和随机森林两种方法在类似的数据上表现得都很好，因此一种常用的方法就是先尝试随机森林，它的鲁棒性很好。如果随机森林效果很好，但预测时间太长，或者机器学习模型精度小数点后第二位的提高也很重要，那么切换成梯度提升通常会有用。

如果你想要将梯度提升应用在大规模问题上，可以研究一下 `xgboost` 包及其 Python 接口，在写作本书时，这个库在许多数据集上的速度都比 `scikit-learn` 对梯度提升的实现要快（有时调参也更简单）。

优点、缺点和参数。梯度提升决策树是监督学习中最强大也最常用的模型之一。其主要缺点是需要仔细调参，而且训练时间可能会比较长。与其他基于树的模型类似，这一算法不需要对数据进行缩放就可以表现得很好，而且也适用于二元特征与连续特征同时存在的数据集。与其他基于树的模型相同，它也通常不适用于高维稀疏数据。

梯度提升树模型的主要参数包括树的数量 `n_estimators` 和学习率 `learning_rate`，后者用于控制每棵树对前一棵树的错误的纠正强度。这两个参数高度相关，因为 `learning_rate` 越低，就需要更多的树来构建具有相似复杂度的模型。随机森林的 `n_estimators` 值总是越大越好，但梯度提升不同，增大 `n_estimators` 会导致模型更加复杂，进而可能导致过拟合。通常的做法是根据时间和内存的预算选择合适的 `n_estimators`，然后对不同的 `learning_rate` 进行遍历。

另一个重要参数是 `max_depth`（或 `max_leaf_nodes`），用于降低每棵树的复杂度。梯度提升模型的 `max_depth` 通常都设置得很小，一般不超过 5。

2.3.7　核支持向量机

我们要讨论的下一种监督学习模型是核支持向量机（kernelized support vector machine）。在 2.3.3 节中，我们研究了将线性支持向量机用于分类任务。核支持向量机（通常简称为 SVM）是可以推广到更复杂模型的扩展，这些模型无法被输入空间的超平面定义。虽然支持向量机可以同时用于分类和回归，但我们只会介绍用于分类的情况，它在 `SVC` 中实现。类似的概念也适用于支持向量回归，后者在 `SVR` 中实现。

核支持向量机背后的数学有点复杂，已经超出了本书的范围。你可以阅读 Hastie、Tibshirani 和 Friedman 合著的《统计学习基础》一书（http://statweb.stanford.edu/~tibs/ElemStatLearn/）的第 12 章了解更多细节。不过，我们会努力向读者传达这一方法背后的理念。

1. 线性模型与非线性特征

如图 2-15 所示，线性模型在低维空间中可能非常受限，因为线和平面的灵活性有限。有一种方法可以让线性模型更加灵活，就是添加更多的特征——举个例子，添加输入特征的交互项或多项式。

我们来看一下 2.3.5 节中用到的模拟数据集（见图 2-29）：

In[76]:

```
X, y = make_blobs(centers=4, random_state=8)
y = y % 2

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

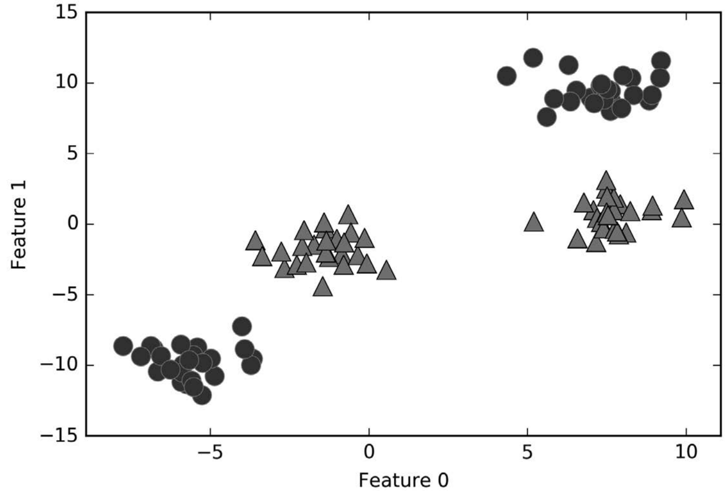


图 2-36：二分类数据集，其类别并不是线性可分的

用于分类的线性模型只能用一条直线来划分数据点，对这个数据集无法给出较好的结果（见图 2-37）：

In[77]:

```
from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

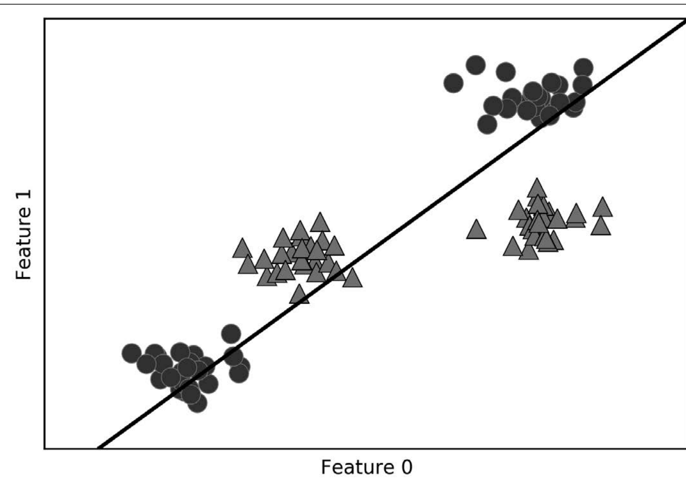


图 2-37：线性 SVM 给出的决策边界

现在我们对输入特征进行扩展，比如说添加第二个特征的平方（feature1 ** 2）作为一个新特征。现在我们将每个数据点表示为三维点 (feature0, feature1, feature1 ** 2)，而不是二维点 (feature0, feature1)[11]。这个新的表示可以画成图 2-38 中的三维散点图：

In[78]:

```
# 添加第二个特征的平方，作为一个新特征
X_new = np.hstack([X, X[:, 1:] ** 2])

from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# 3D可视化
ax = Axes3D(figure, elev=-152, azim=-26)
# 首先画出所有y == 0的点，然后画出所有y == 1的点
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
          cmap=mglearn.cm2, s=60)
ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2")
```

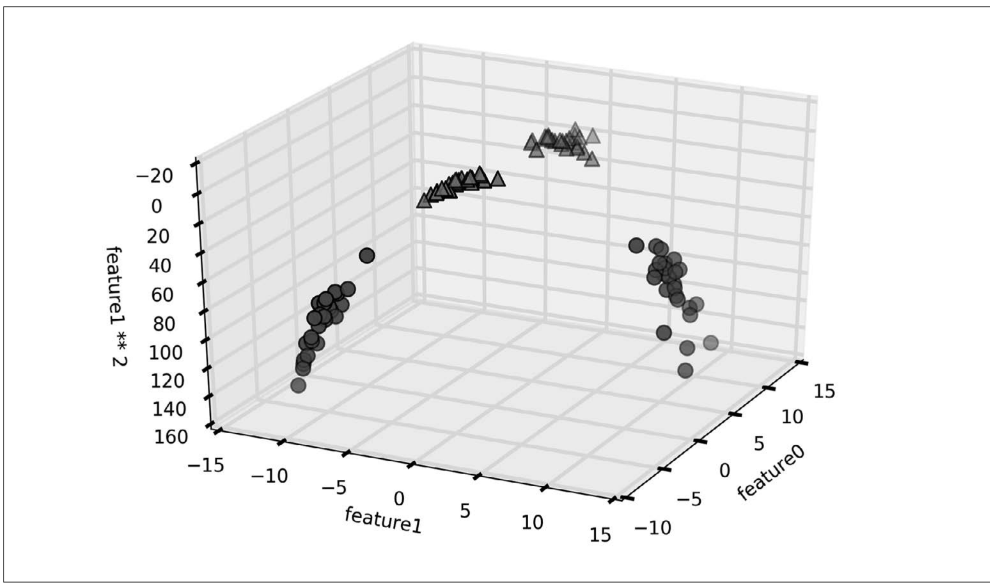


图 2-38：对图 2-37 中的数据集进行扩展，新增由 feature1 导出的的第三个特征

在数据的新表示中，现在可以用线性模型（三维空间中的平面）将这两个类别分开。我们可以用线性模型拟合扩展后的数据来验证这一点（见图 2-39）：

注 11：我们选择添加这个特征只是为了便于说明。这个特征并没有特别的重要性。

In[79]:

```
linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_

# 显示线性决策边界
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)

XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
          cmap=mglearn.cm2, s=60)

ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2")
```

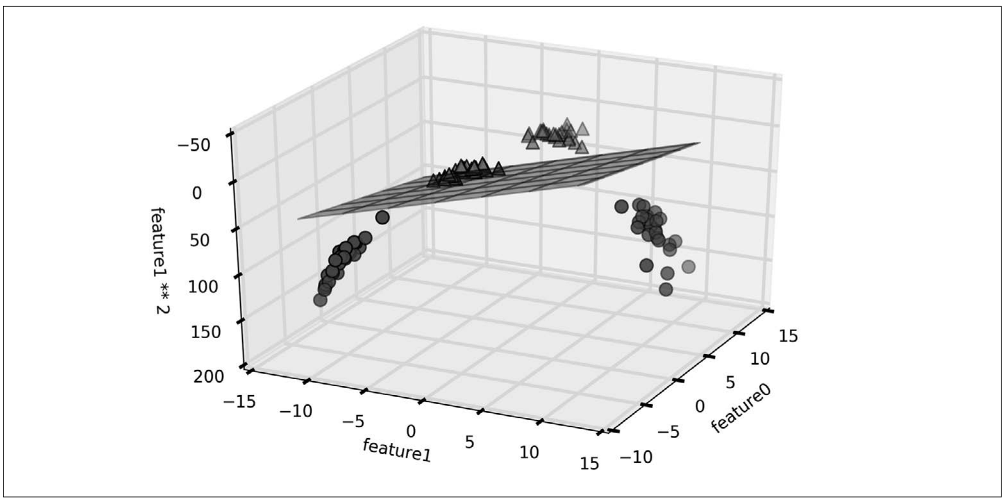


图 2-39：线性 SVM 对扩展后的三维数据集给出的决策边界

如果将线性 SVM 模型看作原始特征的函数，那么它实际上已经不是线性的了。它不是一条直线，而是一个椭圆，你可以在下图中看出（图 2-40）：

In[80]:

```
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
             cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

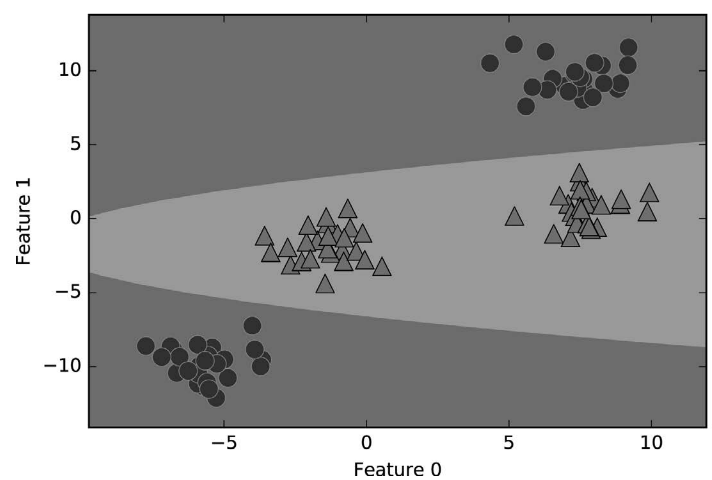


图 2-40：将图 2-39 给出的决策边界作为两个原始特征的函数

2. 核技巧

这里需要记住的是，向数据表示中添加非线性特征，可以让线性模型变得更强大。但是，通常来说我们并不知道要添加哪些特征，而且添加许多特征（比如 100 维特征空间所有可能的交互项）的计算开销可能会很大。幸运的是，有一种巧妙的数学技巧，让我们可以在更高维空间中学习分类器，而不用实际计算可能非常大的新的数据表示。这种技巧叫作**核技巧**（kernel trick），它的原理是直接计算扩展特征表示中数据点之间的距离（更准确地说是内积），而不用实际对扩展进行计算。

对于支持向量机，将数据映射到更高维空间中有两种常用的方法：一种是多项式核，在一定阶数内计算原始特征所有可能的多项式（比如 `feature1 ** 2 * feature2 ** 5`）；另一种是径向基函数（radial basis function，RBF）核，也叫高斯核。高斯核有点难以解释，因为它对应无限维的特征空间。一种对高斯核的解释是它考虑所有阶数的所有可能的多项式，但阶数越高，特征的重要性越小。[12]

不过在实践中，核 SVM 背后的数学细节并不是很重要，可以简单地总结出使用 RBF 核 SVM 进行预测的方法——我们将在下一节介绍这方面的内容。

3. 理解SVM

在训练过程中，SVM 学习每个训练数据点对于表示两个类别之间的决策边界的重要性。通常只有一部分训练数据点对于定义决策边界来说很重要：位于类别之间边界上的那些点。这些点叫作**支持向量**（support vector），支持向量机正是由此得名。

想要对新样本点进行预测，需要测量它与每个支持向量之间的距离。分类决策是基于它与支持向量之间的距离以及在训练过程中学到的支持向量重要性（保存在 `SVC` 的 `dual_coef_` 属性中）来做出的。

注 12：遵循指数映射的泰勒展开。

数据点之间的距离由高斯核给出：

$$k_{\text{rbf}}(x_1, x_2) = \exp(-\gamma \| x_1 - x_2 \|^2)$$

这里 x_1 和 x_2 是数据点，$\| x_1 - x_2 \|$ 表示欧氏距离，γ（gamma）是控制高斯核宽度的参数。

图 2-41 是支持向量机对一个二维二分类数据集的训练结果。决策边界用黑色表示，支持向量是尺寸较大的点。下列代码将在 forge 数据集上训练 SVM 并创建此图：

In[81]:

```
from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
# 画出支持向量
sv = svm.support_vectors_
# 支持向量的类别标签由dual_coef_的正负号给出
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

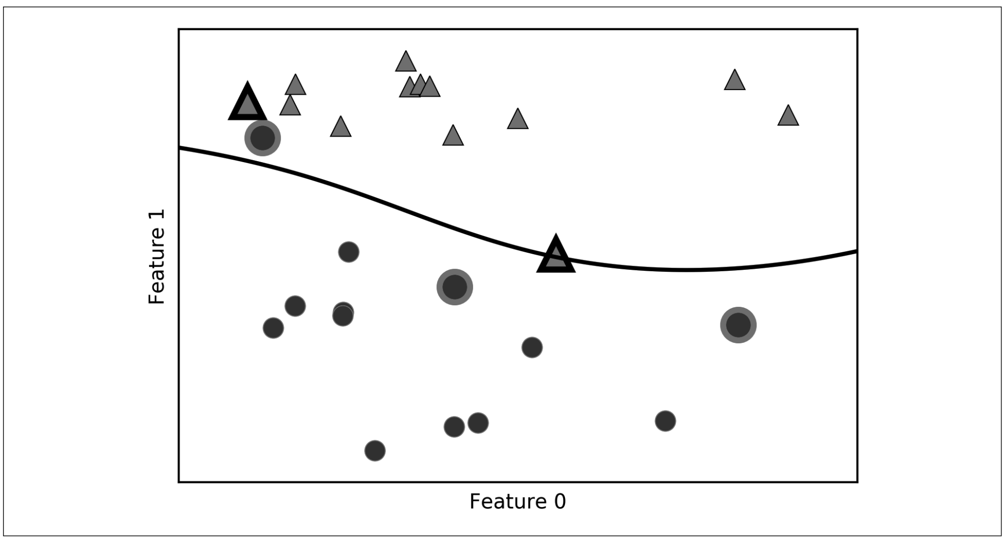


图 2-41：RBF 核 SVM 给出的决策边界和支持向量

在这个例子中，SVM 给出了非常平滑且非线性（不是直线）的边界。这里我们调节了两个参数：C 参数和 gamma 参数，下面我们将详细讨论。

4. SVM调参

gamma 参数是上一节给出的公式中的参数，用于控制高斯核的宽度。它决定了点与点之间"靠近"是指多大的距离。C 参数是正则化参数，与线性模型中用到的类似。它限制每个点的重要性（或者更确切地说，每个点的 dual_coef_）。

我们来看一下，改变这些参数时会发生什么（图 2-42）：

In[82]:
```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"],
                  ncol=4, loc=(.9, 1.2))
```

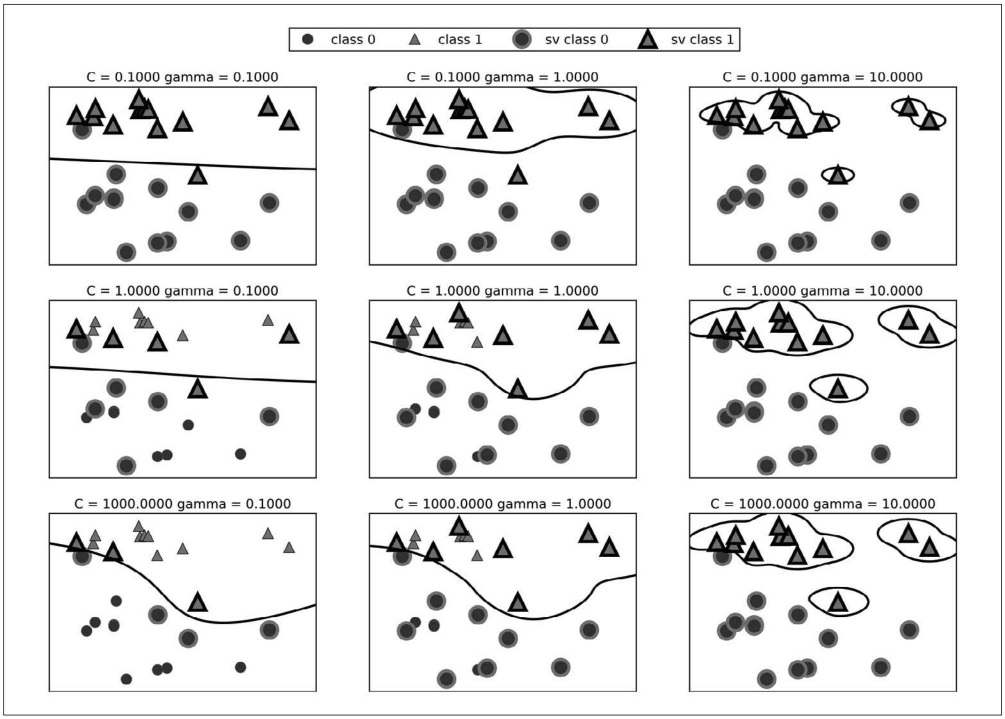


图 2-42：设置不同的 C 和 gamma 参数对应的决策边界和支持向量

从左到右，我们将参数 gamma 的值从 0.1 增加到 10。gamma 较小，说明高斯核的半径较大，许多点都被看作比较靠近。这一点可以在图中看出：左侧的图决策边界非常平滑，越向右的图决策边界更关注单个点。小的 gamma 值表示决策边界变化很慢，生成的是复杂度较低的模型，而大的 gamma 值则会生成更为复杂的模型。

从上到下，我们将参数 C 的值从 0.1 增加到 1000。与线性模型相同，C 值很小，说明模型非常受限，每个数据点的影响范围都有限。你可以看到，左上角的图中，决策边界看起来几乎是线性的，误分类的点对边界几乎没有任何影响。再看左下角的图，增大 C 之后这些点对模型的影响变大，使得决策边界发生弯曲来将这些点正确分类。

我们将 RBF 核 SVM 应用到乳腺癌数据集上。默认情况下，C=1，gamma=1/n_features：

In[83]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

svc = SVC()
svc.fit(X_train, y_train)

print("Accuracy on training set: {:.2f}".format(svc.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(svc.score(X_test, y_test)))
```

Out[83]:

```
Accuracy on training set: 1.00
Accuracy on test set: 0.63
```

这个模型在训练集上的分数十分完美，但在测试集上的精度只有 63%，存在相当严重的过拟合。虽然 SVM 的表现通常都很好，但它对参数的设定和数据的缩放非常敏感。特别地，它要求所有特征有相似的变化范围。我们来看一下每个特征的最小值和最大值，它们绘制在对数坐标上（图 2-43）：

In[84]:

```
plt.plot(X_train.min(axis=0), 'o', label="min")
plt.plot(X_train.max(axis=0), '^', label="max")
plt.legend(loc=4)
plt.xlabel("Feature index")
plt.ylabel("Feature magnitude")
plt.yscale("log")
```

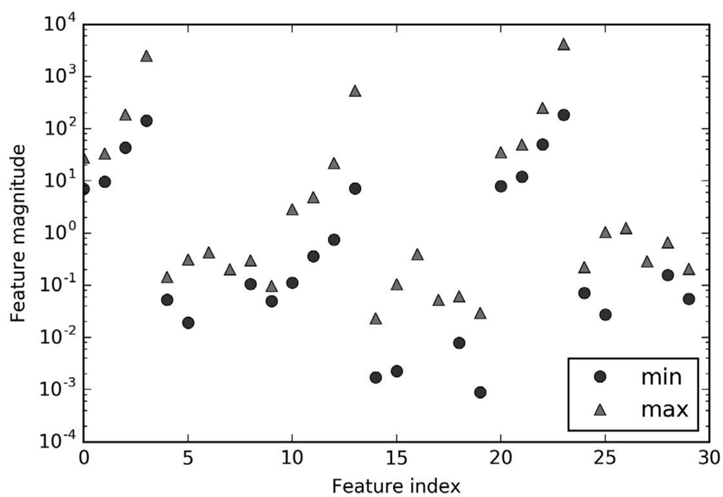


图 2-43：乳腺癌数据集的特征范围（注意 *y* 轴的对数坐标）

从这张图中，我们可以确定乳腺癌数据集的特征具有完全不同的数量级。这对其他模型来说（比如线性模型）可能是小问题，但对核 SVM 却有极大影响。我们来研究处理这个问题的几种方法。

5. 为SVM预处理数据

解决这个问题的一种方法就是对每个特征进行缩放，使其大致都位于同一范围。核 SVM 常用的缩放方法就是将所有特征缩放到 0 和 1 之间。我们将在第 3 章学习如何使用 `MinMaxScaler` 预处理方法来做到这一点，到时会给出更多细节。现在我们来“人工”做到这一点：

In[85]:

```
# 计算训练集中每个特征的最小值
min_on_training = X_train.min(axis=0)
# 计算训练集中每个特征的范围（最大值-最小值）
range_on_training = (X_train - min_on_training).max(axis=0)

# 减去最小值，然后除以范围
# 这样每个特征都是min=0和max=1
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum for each feature\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum for each feature\n {}".format(X_train_scaled.max(axis=0)))
```

Out[85]:

```
Minimum for each feature
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Maximum for each feature
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

In[86]:

```
# 利用训练集的最小值和范围对测试集做相同的变换（详见第3章）
X_test_scaled = (X_test - min_on_training) / range_on_training
```

In[87]:

```
svc = SVC()
svc.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    svc.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Out[87]:

```
Accuracy on training set: 0.948
Accuracy on test set: 0.951
```

数据缩放的作用很大！实际上模型现在处于欠拟合的状态，因为训练集和测试集的性能非常接近，但还没有接近 100% 的精度。从这里开始，我们可以尝试增大 `C` 或 `gamma` 来拟合更为复杂的模型。例如：

In[88]:

```
svc = SVC(C=1000)
svc.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    svc.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Out[88]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.972
```

在这个例子中，增大 C 可以显著改进模型，得到 97.2% 的精度。

6. 优点、缺点和参数

核支持向量机是非常强大的模型，在各种数据集上的表现都很好。SVM 允许决策边界很复杂，即使数据只有几个特征。它在低维数据和高维数据（即很少特征和很多特征）上的表现都很好，但对样本个数的缩放表现不好。在有多达 10 000 个样本的数据上运行 SVM 可能表现良好，但如果数据量达到 100 000 甚至更大，在运行时间和内存使用方面可能会面临挑战。

SVM 的另一个缺点是，预处理数据和调参都需要非常小心。这也是为什么如今很多应用中用的都是基于树的模型，比如随机森林或梯度提升（需要很少的预处理，甚至不需要预处理）。此外，SVM 模型很难检查，可能很难理解为什么会这么预测，而且也难以将模型向非专家进行解释。

不过 SVM 仍然是值得尝试的，特别是所有特征的测量单位相似（比如都是像素密度）而且范围也差不多时。

核 SVM 的重要参数是正则化参数 C、核的选择以及与核相关的参数。虽然我们主要讲的是 RBF 核，但 scikit-learn 中还有其他选择。RBF 核只有一个参数 gamma，它是高斯核宽度的倒数。gamma 和 C 控制的都是模型复杂度，较大的值都对应更为复杂的模型。因此，这两个参数的设定通常是强烈相关的，应该同时调节。

2.3.8 神经网络（深度学习）

一类被称为神经网络的算法最近以“深度学习”的名字再度流行。虽然深度学习在许多机器学习应用中都有巨大的潜力，但深度学习算法往往经过精确调整，只适用于特定的使用场景。这里只讨论一些相对简单的方法，即用于分类和回归的**多层感知机**（multilayer perceptron，MLP），它可以作为研究更复杂的深度学习方法的起点。MLP 也被称为（普通）前馈神经网络，有时也简称为神经网络。

1. 神经网络模型

MLP 可以被视为广义的线性模型，执行多层处理后得到结论。

还记得线性回归的预测公式为：

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \cdots + w[p] * x[p] + b$$

简单来说，$\hat{y}$ 是输入特征 $x[0]$ 到 $x[p]$ 的加权求和，权重为学到的系数 $w[0]$ 到 $w[p]$。我们可以将这个公式可视化，如图 2-44 所示。

In[89]:

```
display(mglearn.plots.plot_logistic_regression_graph())
```

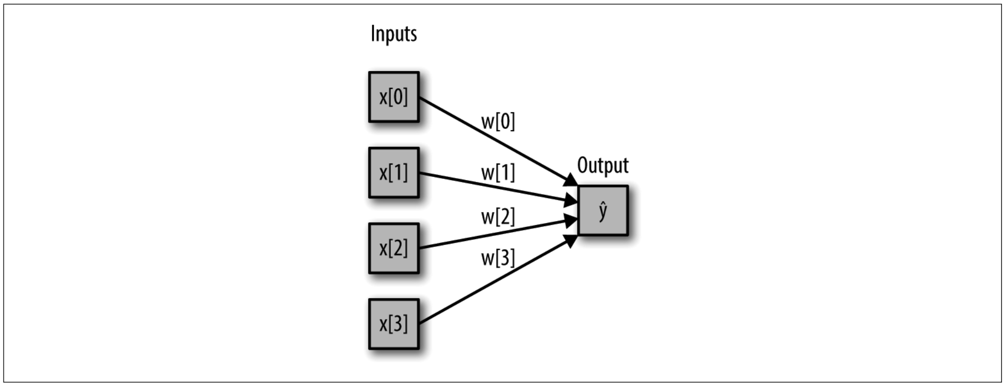


图 2-44：Logistic 回归的可视化，其中输入特征和预测结果显示为结点，系数是结点之间的连线

图中，左边的每个结点代表一个输入特征，连线代表学到的系数，右边的结点代表输出，是输入的加权求和。

在 MLP 中，多次重复这个计算加权求和的过程，首先计算代表中间过程的**隐单元**（hidden unit），然后再计算这些隐单元的加权求和并得到最终结果（如图 2-45 所示）：

In[90]:
```
display(mglearn.plots.plot_single_hidden_layer_graph())
```

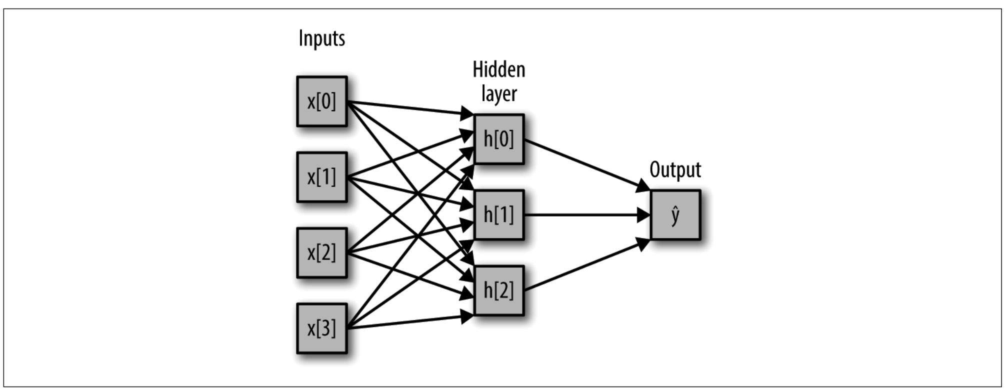


图 2-45：单隐层的多层感知机图示

这个模型需要学习更多的系数（也叫作权重）：在每个输入与每个隐单元（隐单元组成了**隐层**）之间有一个系数，在每个隐单元与输出之间也有一个系数。

从数学的角度看，计算一系列加权求和与只计算一个加权求和是完全相同的，因此，为了让这个模型真正比线性模型更为强大，我们还需要一个技巧。在计算完每个隐单元的加权求和之后，对结果再应用一个非线性函数——通常是**校正非线性**（rectifying nonlinearity，也叫校正线性单元或 relu）或**正切双曲线**（tangens hyperbolicus，tanh）。然后将这个函数的结果用于加权求和，计算得到输出 $\hat{y}$。这两个函数的可视化效果见图 2-46。relu 截断小于

0 的值，而 tanh 在输入值较小时接近 –1，在输入值较大时接近 +1。有了这两种非线性函数，神经网络可以学习比线性模型复杂得多的函数。

In[91]:

```
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("relu(x), tanh(x)")
```

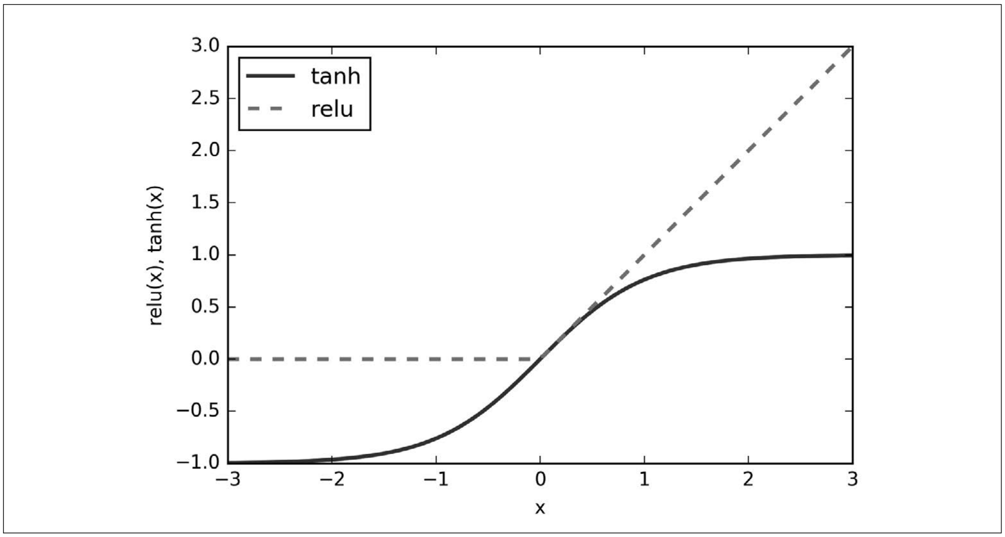


图 2-46：双曲正切激活函数与校正线性激活函数

对于图 2-45 所示的小型神经网络，计算回归问题的 $\hat{y}$ 的完整公式如下（使用 tanh 非线性）：

$$h[0] = \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[0])$$
$$h[1] = \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[1])$$
$$h[2] = \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[2])$$
$$\hat{y} = v[0] * h[0] + v[1] * h[1] + v[2] * h[2] + b$$

其中，w 是输入 x 与隐层 h 之间的权重，v 是隐层 h 与输出 $\hat{y}$ 之间的权重。权重 w 和 v 要从数据中学习得到，x 是输入特征，$\hat{y}$ 是计算得到的输出，h 是计算的中间结果。需要用户设置的一个重要参数是隐层中的结点个数。对于非常小或非常简单的数据集，这个值可以小到 10；对于非常复杂的数据，这个值可以大到 10 000。也可以添加多个隐层，如图 2-47 所示。

In[92]:

```
mglearn.plots.plot_two_hidden_layer_graph()
```

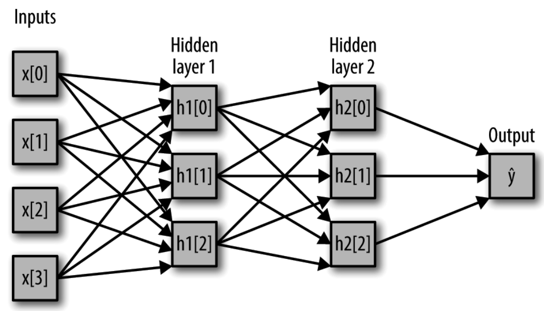


图 2-47：有两个隐层的多层感知机

这些由许多计算层组成的大型神经网络，正是术语“深度学习”的灵感来源。

2. 神经网络调参

我们将 `MLPClassifier` 应用到本章前面用过的 `two_moons` 数据集上，以此研究 MLP 的工作原理。结果如图 2-48 所示。

In[93]:

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

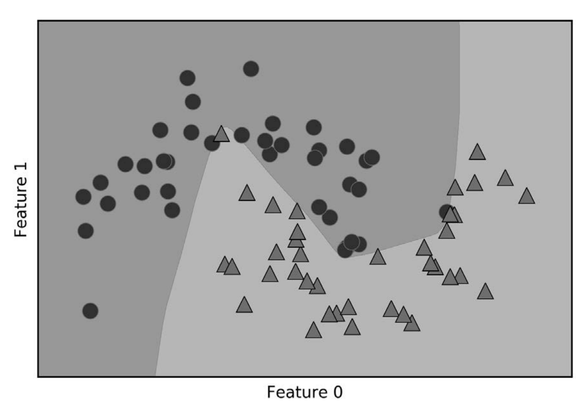


图 2-48：包含 100 个隐单元的神经网络在 `two_moons` 数据集上学到的决策边界

如你所见，神经网络学到的决策边界完全是非线性的，但相对平滑。我们用到了 solver='lbfgs'，这一点稍后会讲到。

默认情况下，MLP 使用 100 个隐结点，这对于这个小型数据集来说已经相当多了。我们可以减少其数量（从而降低了模型复杂度），但仍然得到很好的结果（图 2-49）：

In[94]:

```
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

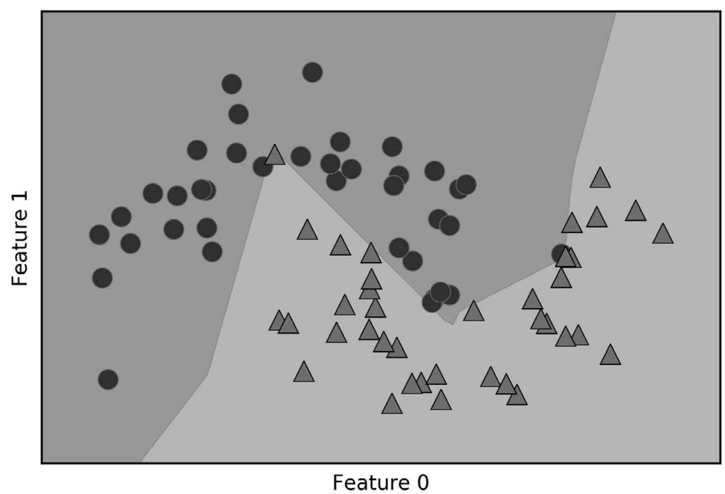


图 2-49：包含 10 个隐单元的神经网络在 two_moons 数据集上学到的决策边界

只有 10 个隐单元时，决策边界看起来更加参差不齐。默认的非线性是 relu，如图 2-46 所示。如果使用单隐层，那么决策函数将由 10 个直线段组成。如果想得到更加平滑的决策边界，可以添加更多的隐单元（见图 2-48）、添加第二个隐层（见图 2-50）或者使用 tanh 非线性（见图 2-51）。

In[95]:

```
# 使用2个隐层，每个包含10个单元
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

In[96]:

```
# 使用2个隐层，每个包含10个单元，这次使用tanh非线性
mlp = MLPClassifier(solver='lbfgs', activation='tanh',
                    random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

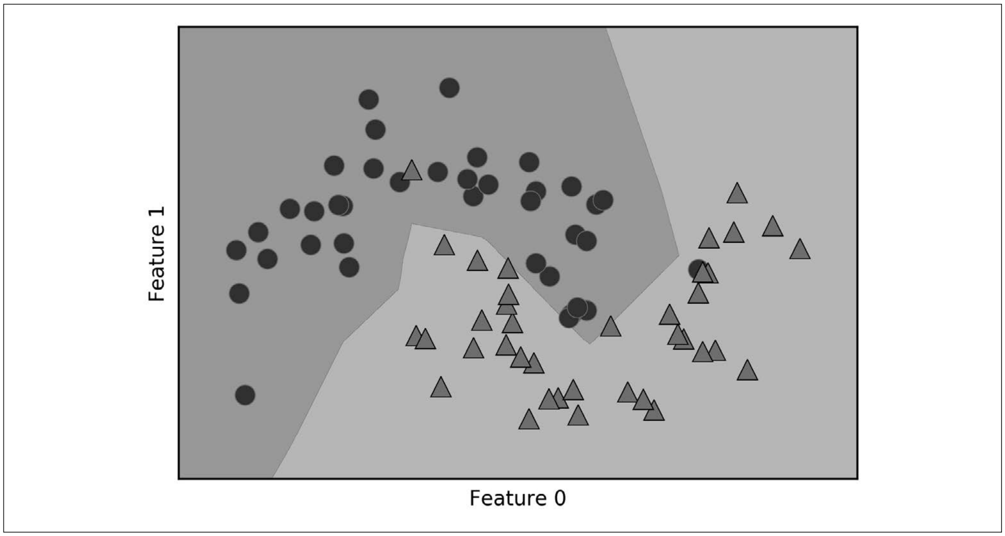


图 2-50：包含 2 个隐层、每个隐层包含 10 个隐单元的神经网络学到的决策边界（激活函数为 relu）

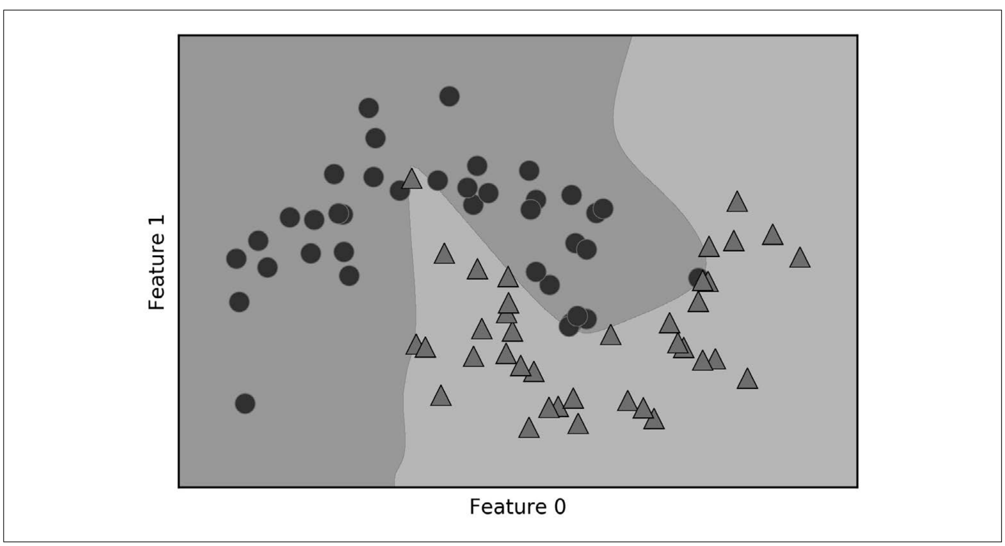


图 2-51：包含 2 个隐层、每个隐层包含 10 个隐单元的神经网络学到的决策边界（激活函数为 tanh）

最后，我们还可以利用 L2 惩罚使权重趋向于 0，从而控制神经网络的复杂度，正如我们在岭回归和线性分类器中所做的那样。MLPClassifier 中调节 L2 惩罚的参数是 alpha（与线性回归模型中的相同），它的默认值很小（弱正则化）。图 2-52 显示了不同 alpha 值对 two_moons 数据集的影响，用的是 2 个隐层的神经网络，每层包含 10 个或 100 个单元：

In[97]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(solver='lbfgs', random_state=0,
                            hidden_layer_sizes=[n_hidden_nodes, n_hidden_nodes],
                            alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
        mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(
                     n_hidden_nodes, n_hidden_nodes, alpha))
```

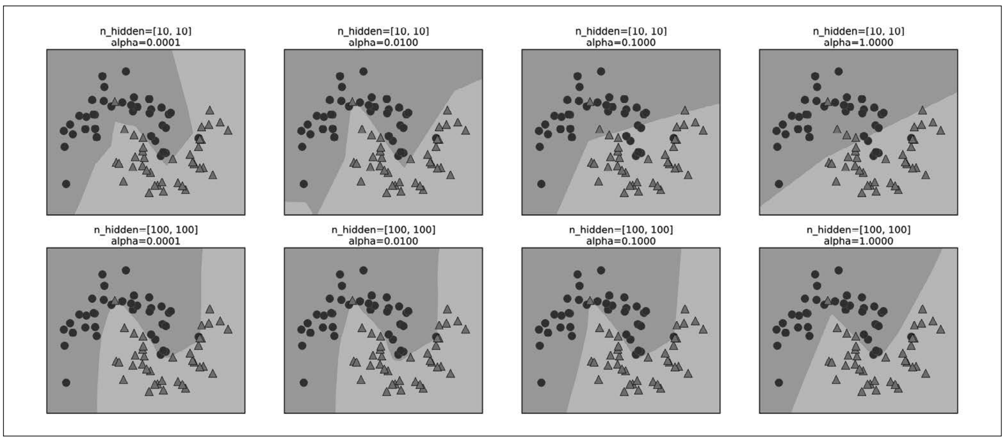


图 2-52：不同隐单元个数与 alpha 参数的不同设定下的决策函数

现在你可能已经认识到了，控制神经网络复杂度的方法有很多种：隐层的个数、每个隐层中的单元个数与正则化（alpha）。实际上还有更多，但这里不再过多介绍。

神经网络的一个重要性质是，在开始学习之前其权重是随机设置的，这种随机初始化会影响学到的模型。也就是说，即使使用完全相同的参数，如果随机种子不同的话，我们也可能得到非常不一样的模型。如果网络很大，并且复杂度选择合理的话，那么这应该不会对精度有太大影响，但应该记住这一点（特别是对于较小的网络）。图 2-53 显示了几个模型的图像，所有模型都使用相同的参数设置进行学习：

In[98]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i,
                        hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
```

```
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
```

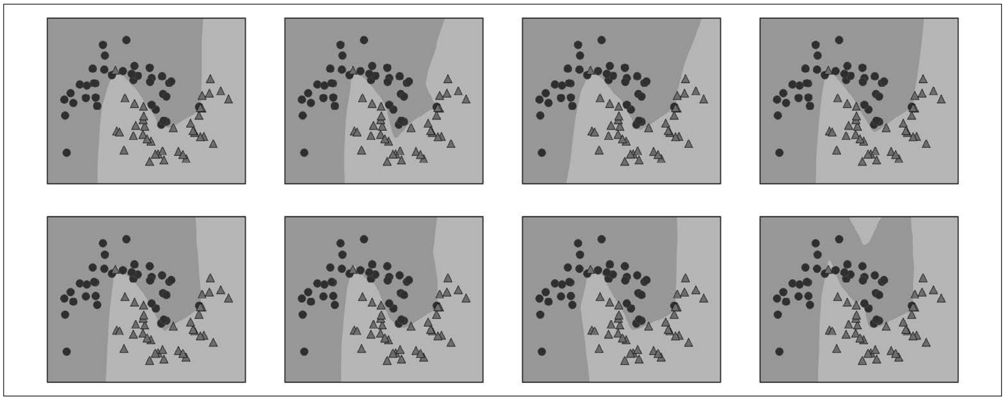


图 2-53：相同参数但不同随机初始化的情况下学到的决策函数

为了在现实世界的数据上进一步理解神经网络，我们将 MLPClassifier 应用在乳腺癌数据集上。首先使用默认参数：

In[99]:

```
print("Cancer data per-feature maxima:\n{}".format(cancer.data.max(axis=0)))
```

Out[99]:

```
Cancer data per-feature maxima:
[   28.110    39.280   188.500  2501.000     0.163     0.345     0.427
     0.201     0.304     0.097     2.873     4.885    21.980   542.200
     0.031     0.135     0.396     0.053     0.079     0.030    36.040
    49.540   251.200  4254.000     0.223     1.058     1.252     0.291
     0.664     0.207]
```

In[100]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Accuracy on training set: {:.2f}".format(mlp.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_test, y_test)))
```

Out[100]:

```
Accuracy on training set: 0.92
Accuracy on test set: 0.90
```

MLP 的精度相当好，但没有其他模型好。与较早的 SVC 例子相同，原因可能在于数据的缩放。神经网络也要求所有输入特征的变化范围相似，最理想的情况是均值为 0、方差为 1。我们必须对数据进行缩放以满足这些要求。同样，我们这里将人工完成，但在第 3 章将会介绍用 StandardScaler 自动完成：

In[101]:

```
# 计算训练集中每个特征的平均值
mean_on_train = X_train.mean(axis=0)
# 计算训练集中每个特征的标准差
std_on_train = X_train.std(axis=0)

# 减去平均值，然后乘以标准差的倒数
# 如此运算之后，mean=0，std=1
X_train_scaled = (X_train - mean_on_train) / std_on_train
# 对测试集做相同的变换（使用训练集的平均值和标准差）
X_test_scaled = (X_test - mean_on_train) / std_on_train

mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[101]:

```
Accuracy on training set: 0.991
Accuracy on test set: 0.965

ConvergenceWarning:
    Stochastic Optimizer: Maximum iterations reached and the optimization
    hasn't converged yet.
```

缩放之后的结果要好得多，而且也相当有竞争力。不过模型给出了一个警告，告诉我们已经达到最大迭代次数。这是用于学习模型的 adam 算法的一部分，告诉我们应该增加迭代次数：

In[102]:

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[102]:

```
Accuracy on training set: 0.995
Accuracy on test set: 0.965
```

增加迭代次数仅提高了训练集性能，但没有提高泛化性能。不过模型的表现相当不错。由于训练性能和测试性能之间仍有一些差距，所以我们可以尝试降低模型复杂度来得到更好的泛化性能。这里我们选择增大 alpha 参数（变化范围相当大，从 0.0001 到 1），以此向权重添加更强的正则化：

In[103]:

```
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
```

```
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[103]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.972
```

这得到了与我们目前最好的模型相同的性能。[13]

虽然可以分析神经网络学到了什么，但这通常比分析线性模型或基于树的模型更为复杂。要想观察模型学到了什么，一种方法是查看模型的权重。你可以在 `scikit-learn` 示例库中查看这样的一个示例（http://scikit-learn.org/stable/auto_examples/neural_networks/plot_mnist_filters.html）。对于乳腺癌数据集，这可能有点难以理解。下面这张图（图 2-54）显示了连接输入和第一个隐层之间的权重。图中的行对应 30 个输入特征，列对应 100 个隐单元。浅色代表较大的正值，而深色代表负值。

In[104]:

```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Columns in weight matrix")
plt.ylabel("Input feature")
plt.colorbar()
```

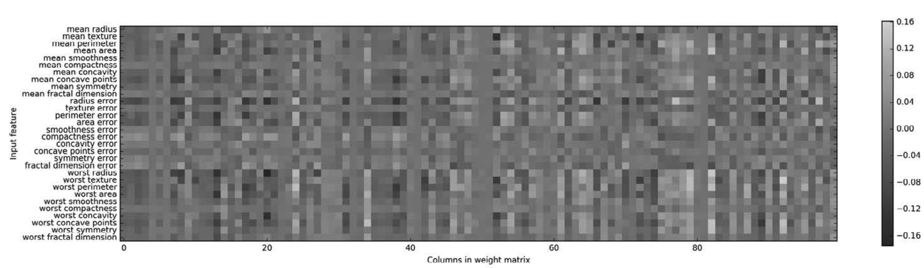


图 2-54：神经网络在乳腺癌数据集上学到的第一个隐层权重的热图

我们可以推断，如果某个特征对所有隐单元的权重都很小，那么这个特征对模型来说就“不太重要”。可以看到，与其他特征相比，“mean smoothness”“mean compactness”以及“smoothness error”和“fractal dimension error”之间的特征的权重都相对较小。这可能说明这些特征不太重要，也可能是我们没有用神经网络可以使用的方式来表示这些特征。

我们还可以将连接隐层和输出层的权重可视化，但它们更加难以解释。

虽然 `MLPClassifier` 和 `MLPRegressor` 为最常见的神经网络架构提供了易于使用的接口，但它们只包含神经网络潜在应用的一部分。如果你有兴趣使用更灵活或更大的模型，我们建

注 13：现在你可能已经注意到了，许多表现很好的模型都得到了完全相同的精度 0.972。这说明所有模型犯错的数量完全相同，也就是 4 个。如果你对比实际预测结果，甚至会发现它们都在相同的地方犯错！这可能是因为数据集非常小，或者是因为这些点与其他点的确不同。

议你看一下除了 `scikit-learn` 之外的很棒的深度学习库。对于 Python 用户来说，最为完善的是 `keras`、`lasagna` 和 `tensor-flow`。`lasagna` 是基于 `theano` 库构建的，而 `keras` 既可以用 `tensor-flow` 也可以用 `theano`。这些库提供了更为灵活的接口，可以用来构建神经网络并跟踪深度学习研究的快速发展。所有流行的深度学习库也都允许使用高性能的图形处理单元（GPU），而 `scikit-learn` 不支持 GPU。使用 GPU 可以将计算速度加快 10 到 100 倍，GPU 对于将深度学习方法应用到大型数据集上至关重要。

3. 优点、缺点和参数

在机器学习的许多应用中，神经网络再次成为最先进的模型。它的主要优点之一是能够获取大量数据中包含的信息，并构建无比复杂的模型。给定足够的计算时间和数据，并且仔细调节参数，神经网络通常可以打败其他机器学习算法（无论是分类任务还是回归任务）。

这就引出了下面要说的缺点。神经网络——特别是功能强大的大型神经网络——通常需要很长的训练时间。它还需要仔细地预处理数据，正如我们这里所看到的。与 SVM 类似，神经网络在“均匀”数据上的性能最好，其中“均匀”是指所有特征都具有相似的含义。如果数据包含不同种类的特征，那么基于树的模型可能表现得更好。神经网络调参本身也是一门艺术。调节神经网络模型和训练模型的方法有很多种，我们只是蜻蜓点水地尝试了几种而已。

估计神经网络的复杂度。最重要的参数是层数和每层的隐单元个数。你应该首先设置 1 个或 2 个隐层，然后可以逐步增加。每个隐层的结点个数通常与输入特征个数接近，但在几千个结点时很少会多于特征个数。

在考虑神经网络的模型复杂度时，一个有用的度量是学到的权重（或系数）的个数。如果你有一个包含 100 个特征的二分类数据集，模型有 100 个隐单元，那么输入层和第一个隐层之间就有 100 * 100 = 10 000 个权重。在隐层和输出层之间还有 100 * 1 = 100 个权重，总共约 10 100 个权重。如果添加含有 100 个隐单元的第二个隐层，那么在第一个隐层和第二个隐层之间又有 100 * 100 = 10 000 个权重，总数变为约 20 100 个权重。如果你使用包含 1000 个隐单元的单隐层，那么在输入层和隐层之间需要学习 100 * 1000 = 100 000 个权重，隐层到输出层之间需要学习 1000 * 1 = 1000 个权重，总共 101 000 个权重。如果再添加第二个隐层，就会增加 1000 * 1000 = 1 000 000 个权重，总数变为巨大的 1 101 000 个权重，这比含有 2 个隐层、每层 100 个单元的模型要大 50 倍。

神经网络调参的常用方法是，首先创建一个大到足以过拟合的网络，确保这个网络可以对任务进行学习。知道训练数据可以被学习之后，要么缩小网络，要么增大 `alpha` 来增强正则化，这可以提高泛化性能。

在我们的实验中，主要关注模型的定义：层数、每层的结点个数、正则化和非线性。这些内容定义了我们想要学习的模型。还有一个问题是，**如何**学习模型或用来学习参数的算法，这一点由 `solver` 参数设定。`solver` 有两个好用的选项。默认选项是 `'adam'`，在大多数情况下效果都很好，但对数据的缩放相当敏感（因此，始终将数据缩放为均值为 0、方差为 1 是很重要的）。另一个选项是 `'lbfgs'`，其鲁棒性相当好，但在大型模型或大型数据集上的时间会比较长。还有更高级的 `'sgd'` 选项，许多深度学习研究人员都会用到。`'sgd'` 选项还有许多其他参数需要调节，以便获得最佳结果。你可以在用户指南中找到所有这些

参数及其定义。当你开始使用 MLP 时，我们建议使用 'adam' 和 'lbfgs'。

> **fit 会重置模型**
>
> scikit-learn 模型的一个重要性质就是，调用 fit 总会重置模型之前学到的所有内容。因此，如果你在一个数据集上构建模型，然后在另一个数据集上再次调用 fit，那么模型会“忘记”从第一个数据集中学到的所有内容。你可以对一个模型多次调用 fit，其结果与在“新”模型上调用 fit 是完全相同的。

2.4　分类器的不确定度估计

我们还没有谈到 scikit-learn 接口的另一个有用之处，就是分类器能够给出预测的不确定度估计。一般来说，你感兴趣的不仅是分类器会预测一个测试点属于哪个类别，还包括它对这个预测的置信程度。在实践中，不同类型的错误会在现实应用中导致非常不同的结果。想象一个用于测试癌症的医疗应用。假阳性预测可能只会让患者接受额外的测试，但假阴性预测却可能导致重病没有得到治疗。第 6 章会进一步探讨这一主题。

scikit-learn 中有两个函数可用于获取分类器的不确定度估计：decision_function 和 predict_proba。大多数分类器（但不是全部）都至少有其中一个函数，很多分类器两个都有。我们来构建一个 GradientBoostingClassifier 分类器（同时拥有 decision_function 和 predict_proba 两个方法），看一下这两个函数对一个模拟的二维数据集的作用：

In[105]:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)

# 为了便于说明，我们将两个类别重命名为"blue"和"red"
y_named = np.array(["blue", "red"])[y]

# 我们可以对任意个数组调用train_test_split
# 所有数组的划分方式都是一致的
X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
    train_test_split(X, y_named, y, random_state=0)

# 构建梯度提升模型
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

2.4.1　决策函数

对于二分类的情况，decision_function 返回值的形状是 (n_samples,)，为每个样本都返回一个浮点数：

In[106]:

```
print("X_test.shape: {}".format(X_test.shape))
print("Decision function shape: {}".format(
    gbrt.decision_function(X_test).shape))
```

Out[106]:

```
X_test.shape: (25, 2)
Decision function shape: (25,)
```

对于类别 1 来说，这个值表示模型对该数据点属于“正”类的置信程度。正值表示对正类的偏好，负值表示对“反类”（其他类）的偏好：

In[107]:

```
# 显示decision_function的前几个元素
print("Decision function:\n{}".format(gbrt.decision_function(X_test)[:6]))
```

Out[107]:

```
Decision function:
[ 4.136 -1.683 -3.951 -3.626  4.29   3.662]
```

我们可以通过仅查看决策函数的正负号来再现预测值：

In[108]:

```
print("Thresholded decision function:\n{}".format(
    gbrt.decision_function(X_test) > 0))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

Out[108]:

```
Thresholded decision function:
[ True False False False  True  True False  True  True  True False  True
  True False  True False False False  True  True  True  True  True False
  False]
Predictions:
['red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red' 'blue'
 'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red' 'red'
 'red' 'blue' 'blue']
```

对于二分类问题，“反”类始终是 classes_ 属性的第一个元素，“正”类是 classes_ 的第二个元素。因此，如果你想要完全再现 predict 的输出，需要利用 classes_ 属性：

In[109]:

```
# 将布尔值True/False转换成0和1
greater_zero = (gbrt.decision_function(X_test) > 0).astype(int)
# 利用0和1作为classes_的索引
pred = gbrt.classes_[greater_zero]
# pred与gbrt.predict的输出完全相同
print("pred is equal to predictions: {}".format(
    np.all(pred == gbrt.predict(X_test))))
```

Out[109]:

```
pred is equal to predictions: True
```

decision_function 可以在任意范围取值，这取决于数据与模型参数：

In[110]:

```
decision_function = gbrt.decision_function(X_test)
print("Decision function minimum: {:.2f} maximum: {:.2f}".format(
    np.min(decision_function), np.max(decision_function)))
```

Out[110]:

```
Decision function minimum: -7.69 maximum: 4.29
```

由于可以任意缩放，因此 decision_function 的输出往往很难解释。

在下面的例子中，我们利用颜色编码在二维平面中画出所有点的 decision_function，还有决策边界，后者我们之前见过。我们将训练点画成圆，将测试数据画成三角（图 2-55）：

In[111]:

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4,
                                fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1],
                                            alpha=.4, cm=mglearn.ReBl)

for ax in axes:
    # 画出训练点和测试点
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Test class 0", "Test class 1", "Train class 0",
                "Train class 1"], ncol=4, loc=(.1, 1.1))
```

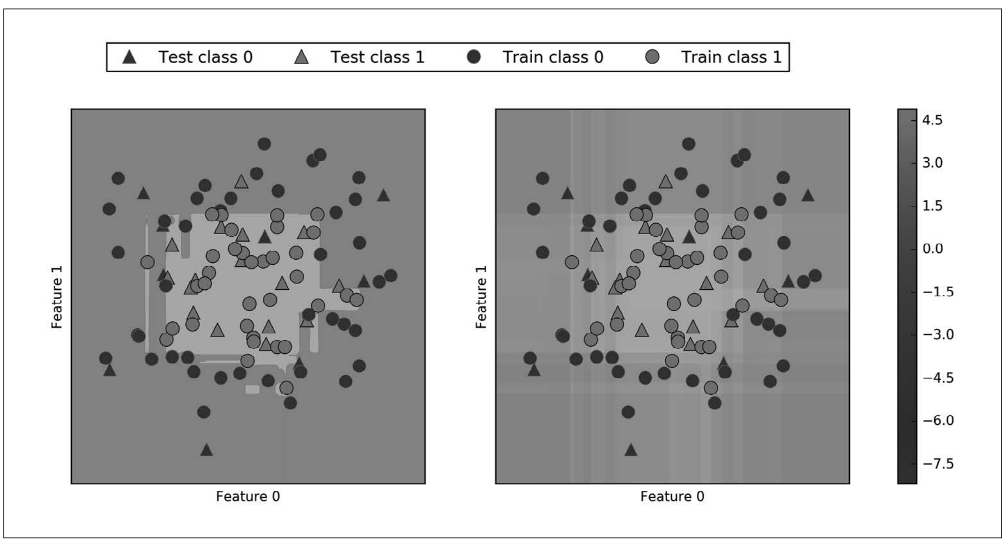


图 2-55：梯度提升模型在一个二维玩具数据集上的决策边界（左）和决策函数（右）

既给出预测结果，又给出分类器的置信程度，这样给出的信息量更大。但在上面的图像中，很难分辨出两个类别之间的边界。

2.4.2 预测概率

predict_proba 的输出是每个类别的概率，通常比 decision_function 的输出更容易理解。对于二分类问题，它的形状始终是 (n_samples, 2)：

In[112]:

```
print("Shape of probabilities: {}".format(gbrt.predict_proba(X_test).shape))
```

Out[112]:

```
Shape of probabilities: (25, 2)
```

每行的第一个元素是第一个类别的估计概率，第二个元素是第二个类别的估计概率。由于 predict_proba 的输出是一个概率，因此总是在 0 和 1 之间，两个类别的元素之和始终为 1：

In[113]:

```
# 显示predict_proba的前几个元素
print("Predicted probabilities:\n{}".format(
    gbrt.predict_proba(X_test[:6])))
```

Out[113]:

```
Predicted probabilities:
[[ 0.016  0.984]
 [ 0.843  0.157]
 [ 0.981  0.019]
 [ 0.974  0.026]
 [ 0.014  0.986]
 [ 0.025  0.975]]
```

由于两个类别的概率之和为 1，因此只有一个类别的概率超过 50%。这个类别就是模型的预测结果。[14]

在上一个输出中可以看到，分类器对大部分点的置信程度都是相对较高的。不确定度大小实际上反映了数据依赖于模型和参数的不确定度。过拟合更强的模型可能会做出置信程度更高的预测，即使可能是错的。复杂度越低的模型通常对预测的不确定度越大。如果模型给出的不确定度符合实际情况，那么这个模型被称为**校正**（calibrated）模型。在校正模型中，如果预测有 70% 的确定度，那么它在 70% 的情况下正确。

在下面的例子中（图 2-56），我们再次给出该数据集的决策边界，以及类别 1 的类别概率：

In[114]:

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot_2d_separator(
    gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(
    gbrt, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl, function='predict_proba')

for ax in axes:
    # 画出训练点和测试点
```

注 14：由于概率是浮点数，所以不太可能两个都等于 0.500。但如果出现了这种情况，预测结果是随机选择的。

```
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Test class 0", "Test class 1", "Train class 0",
                "Train class 1"], ncol=4, loc=(.1, 1.1))
```

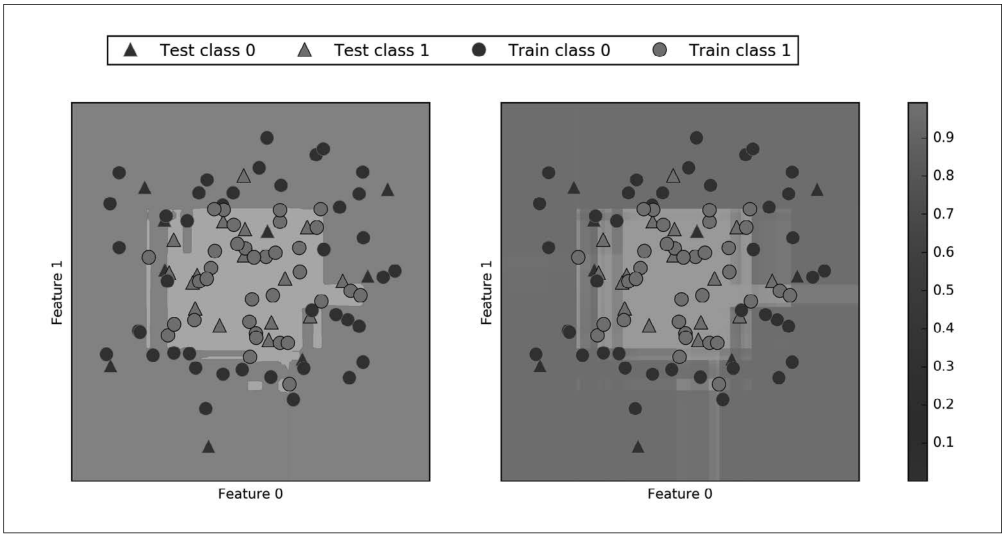


图 2-56：图 2-55 中梯度提升模型的决策边界（左）和预测概率

这张图中的边界更加明确，不确定的小块区域清晰可见。

scikit-learn 网站（http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html）给出了许多模型的对比，以及不确定度估计的形状。我们在图 2-57 中复制了这一结果，我们也建议你去网站查看那些示例。

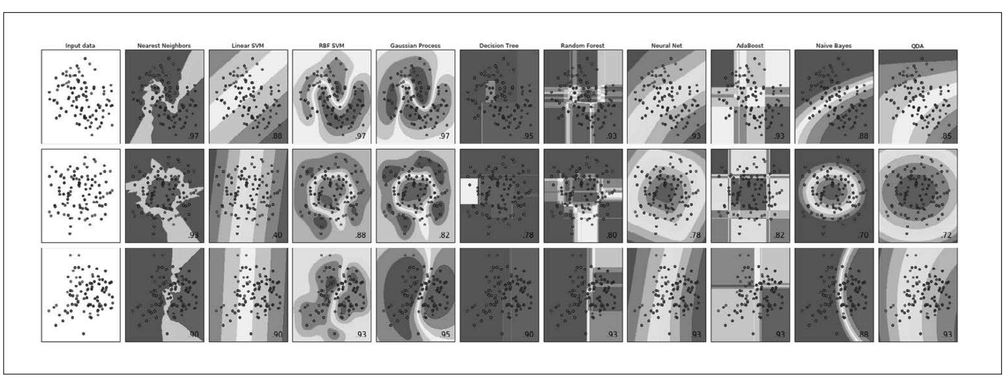


图 2-57：scikit-learn 中几种分类器在模拟数据集上的对比（图片来源：http://scikit-learn.org）

2.4.3 多分类问题的不确定度

到目前为止，我们只讨论了二分类问题中的不确定度估计。但 decision_function 和 predict_proba 也适用于多分类问题。我们将这两个函数应用于鸢尾花（Iris）数据集，这是一个三分类数据集：

In[115]:

```
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

gbrt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbrt.fit(X_train, y_train)
```

In[116]:

```
print("Decision function shape: {}".format(gbrt.decision_function(X_test).shape))
# 显示决策函数的前几个元素
print("Decision function:\n{}".format(gbrt.decision_function(X_test)[:6, :]))
```

Out[116]:

```
Decision function shape: (38, 3)
Decision function:
[[-0.529  1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [-0.524 -0.468  1.52 ]
 [-0.529  1.466 -0.504]
 [-0.531  1.282  0.215]
 [ 1.512 -0.496 -0.503]]
```

对于多分类的情况，decision_function 的形状为 (n_samples, n_classes)，每一列对应每个类别的“确定度分数”，分数较高的类别可能性更大，得分较低的类别可能性较小。你可以找出每个数据点的最大元素，从而利用这些分数再现预测结果：

In[117]:

```
print("Argmax of decision function:\n{}".format(
      np.argmax(gbrt.decision_function(X_test), axis=1)))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

Out[117]:

```
Argmax of decision function:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Predictions:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

predict_proba 输出的形状相同，也是 (n_samples, n_classes)。同样，每个数据点所有可能类别的概率之和为 1：

In[118]:

```
# 显示predict_proba的前几个元素
print("Predicted probabilities:\n{}".format(gbrt.predict_proba(X_test)[:6]))
```

```
# 显示每行的和都是1
print("Sums: {}".format(gbrt.predict_proba(X_test)[:6].sum(axis=1)))
```

Out[118]:

```
Predicted probabilities:
[[ 0.107  0.784  0.109]
 [ 0.789  0.106  0.105]
 [ 0.102  0.108  0.789]
 [ 0.107  0.784  0.109]
 [ 0.108  0.663  0.228]
 [ 0.789  0.106  0.105]]
Sums: [ 1.  1.  1.  1.  1.  1.]
```

同样，我们可以通过计算 predict_proba 的 argmax 来再现预测结果：

In[119]:

```
print("Argmax of predicted probabilities:\n{}".format(
      np.argmax(gbrt.predict_proba(X_test), axis=1)))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

Out[119]:

```
Argmax of predicted probabilities:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Predictions:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

总之，predict_proba 和 decision_function 的形状始终相同，都是 (n_samples, n_classes)——除了二分类特殊情况下的 decision_function。对于二分类的情况，decision_function 只有一列，对应“正”类 classes_[1]。这主要是由于历史原因。

如果有 n_classes 列，你可以通过计算每一列的 argmax 来再现预测结果。但如果类别是字符串，或者是整数，但不是从 0 开始的连续整数的话，一定要小心。如果你想要对比 predict 的结果与 decision_function 或 predict_proba 的结果，一定要用分类器的 classes_ 属性来获取真实的属性名称：

In[120]:

```
logreg = LogisticRegression()

# 用Iris数据集的类别名称来表示每一个目标值
named_target = iris.target_names[y_train]
logreg.fit(X_train, named_target)
print("unique classes in training data: {}".format(logreg.classes_))
print("predictions: {}".format(logreg.predict(X_test)[:10]))
argmax_dec_func = np.argmax(logreg.decision_function(X_test), axis=1)
print("argmax of decision function: {}".format(argmax_dec_func[:10]))
print("argmax combined with classes_: {}".format(
        logreg.classes_[argmax_dec_func][:10]))
```

Out[120]:

```
unique classes in training data: ['setosa' 'versicolor' 'virginica']
predictions: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor'
 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
```

```
argmax of decision function: [1 0 2 1 1 0 1 2 1 1]
argmax combined with classes_: ['versicolor' 'setosa' 'virginica' 'versicolor'
 'versicolor' 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
```

2.5 小结与展望

本章首先讨论了模型复杂度，然后讨论了**泛化**，或者说学习一个能够在前所未见的新数据上表现良好的模型。这就引出了欠拟合和过拟合的概念，前者是指一个模型无法获取训练数据中的所有变化，后者是指模型过分关注训练数据，但对新数据的泛化性能不好。

然后本章讨论了一系列用于分类和回归的机器学习模型，各个模型的优点和缺点，以及如何控制它们的模型复杂度。我们发现，对于许多算法而言，设置正确的参数对模型性能至关重要。有些算法还对输入数据的表示方式很敏感，特别是特征的缩放。因此，如果盲目地将一个算法应用于数据集，而不去理解模型所做的假设以及参数设定的含义，不太可能会得到精度高的模型。

本章包含大量有关算法的信息，在继续阅读后续章节之前你不必记住所有这些细节。但是，这里提到的有关模型的某些知识（以及在特定情况下使用哪种模型）对于在实践中成功应用机器学习模型是很重要的。关于何时使用哪种模型，下面是一份快速总结。

最近邻
适用于小型数据集，是很好的基准模型，很容易解释。

线性模型
非常可靠的首选算法，适用于非常大的数据集，也适用于高维数据。

朴素贝叶斯
只适用于分类问题。比线性模型速度还快，适用于非常大的数据集和高维数据。精度通常要低于线性模型。

决策树
速度很快，不需要数据缩放，可以可视化，很容易解释。

随机森林
几乎总是比单棵决策树的表现要好，鲁棒性很好，非常强大。不需要数据缩放。不适用于高维稀疏数据。

梯度提升决策树
精度通常比随机森林略高。与随机森林相比，训练速度更慢，但预测速度更快，需要的内存也更少。比随机森林需要更多的参数调节。

支持向量机
对于特征含义相似的中等大小的数据集很强大。需要数据缩放，对参数敏感。

神经网络
可以构建非常复杂的模型，特别是对于大型数据集而言。对数据缩放敏感，对参数选取敏感。大型网络需要很长的训练时间。

面对新数据集，通常最好先从简单模型开始，比如线性模型、朴素贝叶斯或最近邻分类器，看能得到什么样的结果。对数据有了进一步了解之后，你可以考虑用于构建更复杂模型的算法，比如随机森林、梯度提升决策树、SVM 或神经网络。

现在你应该对如何应用、调节和分析我们介绍过的模型有了一定的了解。本章主要介绍了二分类问题，因为这通常是最容易理解的。不过本章大多数算法都可以同时用于分类和回归，而且所有分类算法都可以同时用于二分类和多分类。你可以尝试将这些算法应用于 `scikit-learn` 的内置数据集，比如用于回归的 `boston_housing` 或 `diabetes` 数据集，或者用于多分类的 `digits` 数据集。在不同的数据集上实验这些算法，可以让你更好地感受它们所需的训练时间、分析模型的难易程度以及它们对数据表示的敏感程度。

虽然我们分析了不同的参数设定对算法的影响，但在生产环境中实际构建一个对新数据泛化性能很好的模型要更复杂一些。我们将在第 6 章介绍正确调参的方法和自动寻找最佳参数的方法。

不过首先，我们将在下一章深入讨论无监督学习和预处理。

第 3 章

无监督学习与预处理

我们要讨论的第二种机器学习算法是无监督学习算法。无监督学习包括没有已知输出、没有老师指导学习算法的各种机器学习。在无监督学习中，学习算法只有输入数据，并需要从这些数据中提取知识。

3.1 无监督学习的类型

本章将研究两种类型的无监督学习：数据集变换与聚类。

数据集的**无监督变换**（unsupervised transformation）是创建数据新的表示的算法，与数据的原始表示相比，新的表示可能更容易被人或其他机器学习算法所理解。无监督变换的一个常见应用是降维（dimensionality reduction），它接受包含许多特征的数据的高维表示，并找到表示该数据的一种新方法，用较少的特征就可以概括其重要特性。降维的一个常见应用是为了可视化将数据降为二维。

无监督变换的另一个应用是找到“构成”数据的各个组成部分。这方面的一个例子就是对文本文档集合进行主题提取。这里的任务是找到每个文档中讨论的未知主题，并学习每个文档中出现了哪些主题。这可以用于追踪社交媒体上的话题讨论，比如选举、枪支管制或流行歌手等话题。

与之相反，**聚类算法**（clustering algorithm）将数据划分成不同的组，每组包含相似的物项。思考向社交媒体网站上传照片的例子。为了方便你整理照片，网站可能想要将同一个人的照片分在一组。但网站并不知道每张照片是谁，也不知道你的照片集中出现了多少个人。明智的做法是提取所有的人脸，并将看起来相似的人脸分在一组。但愿这些人脸对应同一个人，这样图片的分组也就完成了。

3.2　无监督学习的挑战

无监督学习的一个主要挑战就是评估算法是否学到了有用的东西。无监督学习算法一般用于不包含任何标签信息的数据，所以我们不知道正确的输出应该是什么。因此很难判断一个模型是否“表现很好”。例如，假设我们的聚类算法已经将所有的侧脸照片和所有的正面照片进行分组。这肯定是人脸照片集合的一种可能的划分方法，但并不是我们想要的那种方法。然而，我们没有办法“告诉”算法我们要的是什么，通常来说，评估无监督算法结果的唯一方法就是人工检查。

因此，如果数据科学家想要更好地理解数据，那么无监督算法通常可用于探索性的目的，而不是作为大型自动化系统的一部分。无监督算法的另一个常见应用是作为监督算法的预处理步骤。学习数据的一种新表示，有时可以提高监督算法的精度，或者可以减少内存占用和时间开销。

在开始学习“真正的”无监督算法之前，我们先简要讨论几种简单又常用的预处理方法。虽然预处理和缩放通常与监督学习算法一起使用，但缩放方法并没有用到与“监督”有关的信息，所以它是无监督的。

3.3　预处理与缩放

上一章我们学到，一些算法（如神经网络和 SVM）对数据缩放非常敏感。因此，通常的做法是对特征进行调节，使数据表示更适合于这些算法。通常来说，这是对数据的一种简单的按特征的缩放和移动。下面的代码（图 3-1）给出了一个简单的例子：

In[2]:

```
mglearn.plots.plot_scaling()
```

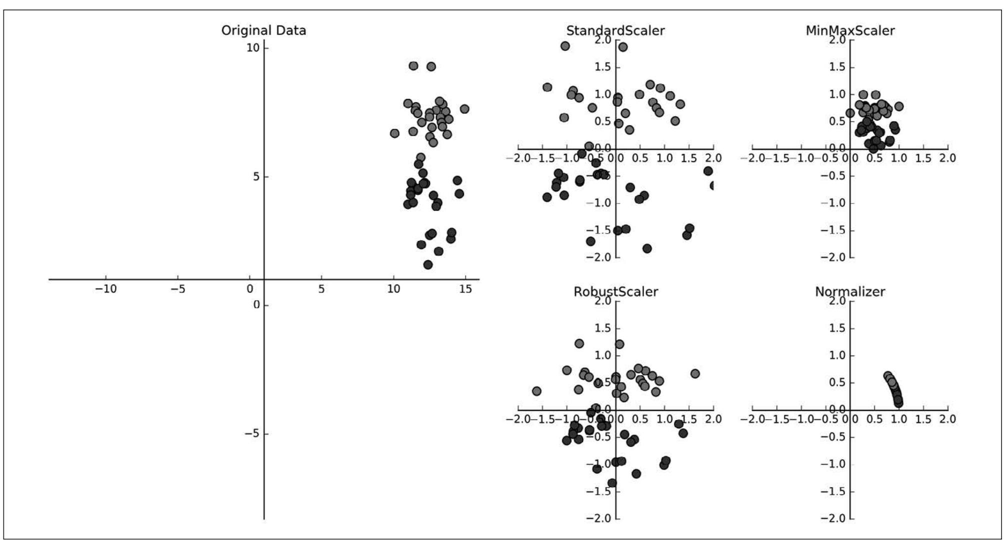


图 3-1：对数据集缩放和预处理的各种方法

3.3.1 不同类型的预处理

在图 3-1 中，第一张图显示的是一个模拟的有两个特征的二分类数据集。第一个特征（x 轴）位于 10 到 15 之间。第二个特征（y 轴）大约位于 1 到 9 之间。

接下来的 4 张图展示了 4 种数据变换方法，都生成了更加标准的范围。scikit-learn 中的 StandardScaler 确保每个特征的平均值为 0、方差为 1，使所有特征都位于同一量级。但这种缩放不能保证特征任何特定的最大值和最小值。RobustScaler 的工作原理与 StandardScaler 类似，确保每个特征的统计属性都位于同一范围。但 RobustScaler 使用的是中位数和四分位数[1]，而不是平均值和方差。这样 RobustScaler 会忽略与其他点有很大不同的数据点（比如测量误差）。这些与众不同的数据点也叫**异常值**（outlier），可能会给其他缩放方法造成麻烦。

与之相反，MinMaxScaler 移动数据，使所有特征都刚好位于 0 到 1 之间。对于二维数据集来说，所有的数据都包含在 x 轴 0 到 1 与 y 轴 0 到 1 组成的矩形中。

最后，Normalizer 用到一种完全不同的缩放方法。它对每个数据点进行缩放，使得特征向量的欧式长度等于 1。换句话说，它将一个数据点投射到半径为 1 的圆上（对于更高维度的情况，是球面）。这意味着每个数据点的缩放比例都不相同（乘以其长度的倒数）。如果只有数据的方向（或角度）是重要的，而特征向量的长度无关紧要，那么通常会使用这种归一化。

3.3.2 应用数据变换

前面我们已经看到不同类型的变换的作用，下面利用 scikit-learn 来应用这些变换。我们将使用第 2 章见过的 cancer 数据集。通常在应用监督学习算法之前使用预处理方法（比如缩放）。举个例子，比如我们想要将核 SVM（SVC）应用在 cancer 数据集上，并使用 MinMaxScaler 来预处理数据。首先加载数据集并将其分为训练集和测试集（我们需要分开的训练集和数据集来对预处理后构建的监督模型进行评估）：

In[3]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=1)
print(X_train.shape)
print(X_test.shape)
```

Out[3]:

```
(426, 30)
(143, 30)
```

注 1：对于一组数字来说，中位数指的是这样的数值 x：有一半数值小于 x，另一半数值大于 x。较小四分位数指的是这样的数值 x：有四分之一的数值小于 x。较大四分位数指的是这样的数值 x：有四分之一的数值大于 x。

提醒一下，这个数据集包含 569 个数据点，每个数据点由 30 个测量值表示。我们将数据集分成包含 426 个样本的训练集与包含 143 个样本的测试集。

与之前构建的监督模型一样，我们首先导入实现预处理的类，然后将其实例化：

In[4]:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

然后，使用 `fit` 方法拟合缩放器（scaler），并将其应用于训练数据。对于 `MinMaxScaler` 来说，`fit` 方法计算训练集中每个特征的最大值和最小值。与第 2 章中的分类器和回归器（regressor）不同，在对缩放器调用 `fit` 时只提供了 `X_train`，而不用 `y_train`：

In[5]:

```
scaler.fit(X_train)
```

Out[5]:

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

为了应用刚刚学习的变换（即对训练数据进行实际**缩放**），我们使用缩放器的 `transform` 方法。在 `scikit-learn` 中，每当模型返回数据的一种新表示时，都可以使用 `transform` 方法：

In[6]:

```
# 变换数据
X_train_scaled = scaler.transform(X_train)
# 在缩放之前和之后分别打印数据集属性
print("transformed shape: {}".format(X_train_scaled.shape))
print("per-feature minimum before scaling:\n {}".format(X_train.min(axis=0)))
print("per-feature maximum before scaling:\n {}".format(X_train.max(axis=0)))
print("per-feature minimum after scaling:\n {}".format(
    X_train_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n {}".format(
    X_train_scaled.max(axis=0)))
```

Out[6]:

```
transformed shape: (426, 30)
per-feature minimum before scaling:
 [   6.98    9.71   43.79  143.50    0.05    0.02    0.      0.      0.11
     0.05    0.12    0.36    0.76    6.80    0.      0.      0.      0.
     0.01    0.      7.93   12.02   50.41  185.20    0.07    0.03    0.
     0.      0.16    0.06]
per-feature maximum before scaling:
[   28.11    39.28   188.5   2501.0     0.16     0.29     0.43     0.2
     0.300    0.100    2.87     4.88    21.98   542.20     0.03     0.14
     0.400    0.050    0.06     0.03    36.04    49.54   251.20  4254.00
     0.220    0.940    1.17     0.29     0.58     0.15]
per-feature minimum after scaling:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
per-feature maximum after scaling:
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

变换后的数据形状与原始数据相同，特征只是发生了移动和缩放。你可以看到，现在所有特征都位于 0 到 1 之间，这也符合我们的预期。

为了将 SVM 应用到缩放后的数据上，还需要对测试集进行变换。这可以通过对 `X_test` 调用 `transform` 方法来完成：

In[7]:

```
# 对测试数据进行变换
X_test_scaled = scaler.transform(X_test)
# 在缩放之后打印测试数据的属性
print("per-feature minimum after scaling:\n{}".format(X_test_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n{}".format(X_test_scaled.max(axis=0)))
```

Out[7]:

```
per-feature minimum after scaling:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.     0.     0.154 -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.     0.    -0.032  0.007
  0.027  0.058  0.02   0.009  0.109  0.026  0.     0.    -0.    -0.002]
per-feature maximum after scaling:
[ 0.958  0.815  0.956  0.894  0.811  1.22   0.88   0.933  0.932  1.037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337  0.391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07   0.924  1.205  1.631]
```

你可以发现，对测试集缩放后的最大值和最小值不是 1 和 0，这或许有些出乎意料。有些特征甚至在 0~1 的范围之外！对此的解释是，`MinMaxScaler`（以及其他所有缩放器）总是对训练集和测试集应用完全相同的变换。也就是说，`transform` 方法总是减去训练集的最小值，然后除以训练集的范围，而这两个值可能与测试集的最小值和范围并不相同。

3.3.3　对训练数据和测试数据进行相同的缩放

为了让监督模型能够在测试集上运行，对训练集和测试集应用完全相同的变换是很重要的。如果我们使用测试集的最小值和范围，下面这个例子（图 3-2）展示了会发生什么：

In[8]:

```
from sklearn.datasets import make_blobs
# 构造数据
X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
# 将其分为训练集和测试集
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)

# 绘制训练集和测试集
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
axes[0].scatter(X_train[:, 0], X_train[:, 1],
                c=mglearn.cm2(0), label="Training set", s=60)
axes[0].scatter(X_test[:, 0], X_test[:, 1], marker='^',
                c=mglearn.cm2(1), label="Test set", s=60)
axes[0].legend(loc='upper left')
axes[0].set_title("Original Data")

# 利用MinMaxScaler缩放数据
scaler = MinMaxScaler()
```

```
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 将正确缩放的数据可视化
axes[1].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mglearn.cm2(0), label="Training set", s=60)
axes[1].scatter(X_test_scaled[:, 0], X_test_scaled[:, 1], marker='^',
                c=mglearn.cm2(1), label="Test set", s=60)
axes[1].set_title("Scaled Data")

# 单独对测试集进行缩放
# 使得测试集的最小值为0，最大值为1
# 千万不要这么做！这里只是为了举例
test_scaler = MinMaxScaler()
test_scaler.fit(X_test)
X_test_scaled_badly = test_scaler.transform(X_test)

# 将错误缩放的数据可视化
axes[2].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mglearn.cm2(0), label="training set", s=60)
axes[2].scatter(X_test_scaled_badly[:, 0], X_test_scaled_badly[:, 1],
                marker='^', c=mglearn.cm2(1), label="test set", s=60)
axes[2].set_title("Improperly Scaled Data")

for ax in axes:
ax.set_xlabel("Feature 0")
ax.set_ylabel("Feature 1")
```

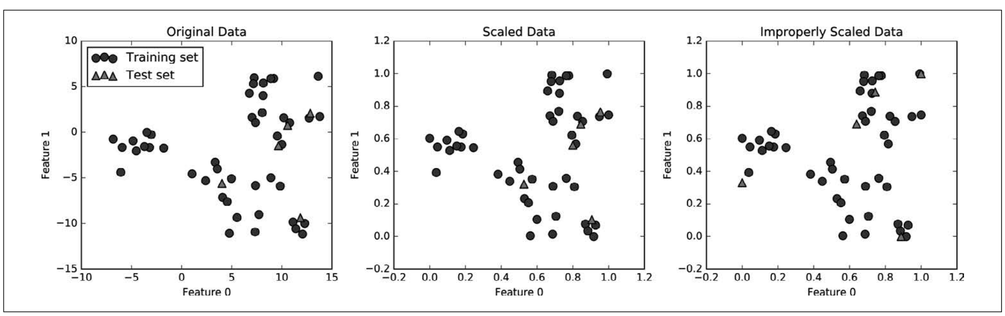


图 3-2：对左图中的训练数据和测试数据同时缩放的效果（中）和分别缩放的效果（右）

第一张图是未缩放的二维数据集，其中训练集用圆形表示，测试集用三角形表示。第二张图中是同样的数据，但使用 MinMaxScaler 缩放。这里我们调用 fit 作用在训练集上，然后调用 transform 作用在训练集和测试集上。你可以发现，第二张图中的数据集看起来与第一张图中的完全相同，只是坐标轴刻度发生了变化。现在所有特征都位于 0 到 1 之间。你还可以发现，测试数据（三角形）的特征最大值和最小值并不是 1 和 0。

第三张图展示了如果我们对训练集和测试集分别进行缩放会发生什么。在这种情况下，对训练集和测试集而言，特征的最大值和最小值都是 1 和 0。但现在数据集看起来不一样。测试集相对训练集的移动不一致，因为它们分别做了不同的缩放。我们随意改变了数据的

排列。这显然不是我们想要做的事情。

再换一种思考方式，想象你的测试集只有一个点。对于一个点而言，无法将其正确地缩放以满足 MinMaxScaler 的最大值和最小值的要求。但是，测试集的大小不应该对你的处理方式有影响。

快捷方式与高效的替代方法

通常来说，你想要在某个数据集上 fit 一个模型，然后再将其 transform。这是一个非常常见的任务，通常可以用比先调用 fit 再调用 transform 更高效的方法来计算。对于这种使用场景，所有具有 transform 方法的模型也都具有一个 fit_transform 方法。下面是使用 StandardScaler 的一个例子：

In[9]:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# 依次调用fit和transform（使用方法链）
X_scaled = scaler.fit(X).transform(X)
# 结果相同，但计算更加高效
X_scaled_d = scaler.fit_transform(X)
```

虽然 fit_transform 不一定对所有模型都更加高效，但在尝试变换训练集时，使用这一方法仍然是很好的做法。

3.3.4 预处理对监督学习的作用

现在我们回到 cancer 数据集，观察使用 MinMaxScaler 对学习 SVC 的作用（这是一种不同的方法，实现了与第 2 章中相同的缩放）。首先，为了对比，我们再次在原始数据上拟合 SVC：

In[10]:

```
from sklearn.svm import SVC

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=0)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Test set accuracy: {:.2f}".format(svm.score(X_test, y_test)))
```

Out[10]:

```
Test set accuracy: 0.63
```

下面先用 MinMaxScaler 对数据进行缩放，然后再拟合 SVC：

In[11]:

```
# 使用0-1缩放进行预处理
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# 在缩放后的训练数据上学习SVM
svm.fit(X_train_scaled, y_train)

# 在缩放后的测试集上计算分数
print("Scaled test set accuracy: {:.2f}".format(
    svm.score(X_test_scaled, y_test)))
```

Out[11]:

```
Scaled test set accuracy: 0.97
```

正如我们上面所见，数据缩放的作用非常显著。虽然数据缩放不涉及任何复杂的数学，但良好的做法仍然是使用 scikit-learn 提供的缩放机制，而不是自己重新实现它们，因为即使在这些简单的计算中也容易犯错。

你也可以通过改变使用的类将一种预处理算法轻松替换成另一种，因为所有的预处理类都具有相同的接口，都包含 fit 和 transform 方法：

In[12]:

```
# 利用零均值和单位方差的缩放方法进行预处理
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 在缩放后的训练数据上学习SVM
svm.fit(X_train_scaled, y_train)

# 在缩放后的测试集上计算分数
print("SVM test accuracy: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

Out[12]:

```
SVM test accuracy: 0.96
```

前面我们已经看到了用于预处理的简单数据变换的工作原理，下面继续学习利用无监督学习进行更有趣的变换。

3.4 降维、特征提取与流形学习

前面讨论过，利用无监督学习进行数据变换可能有很多种目的。最常见的目的就是可视化、压缩数据，以及寻找信息量更大的数据表示以用于进一步的处理。

为了实现这些目的，最简单也最常用的一种算法就是主成分分析。我们也将学习另外两种算法：非负矩阵分解（NMF）和 t-SNE，前者通常用于特征提取，后者通常用于二维散点图的可视化。

3.4.1 主成分分析

主成分分析（principal component analysis，PCA）是一种旋转数据集的方法，旋转后的特征在统计上不相关。在做完这种旋转之后，通常是根据新特征对解释数据的重要性来选择

它的一个子集。下面的例子（图 3-3）展示了 PCA 对一个模拟二维数据集的作用：

In[13]:
```
mglearn.plots.plot_pca_illustration()
```

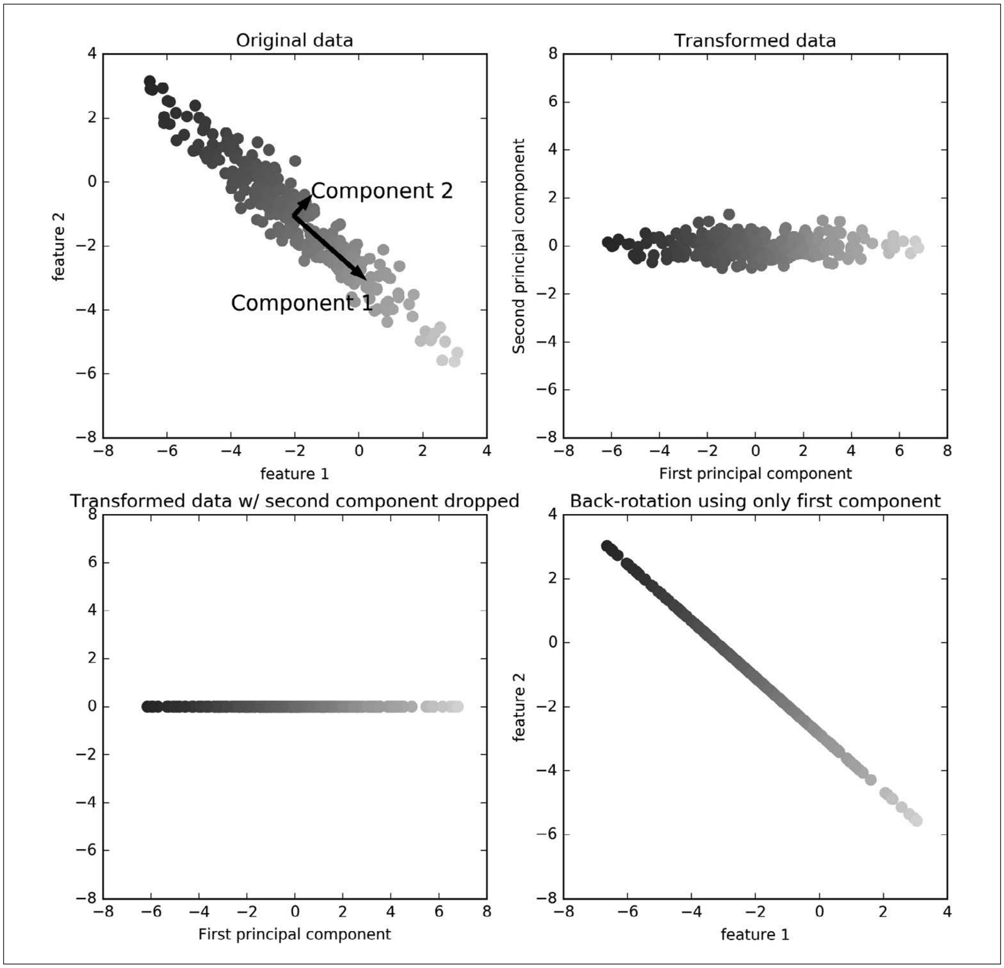


图 3-3：用 PCA 做数据变换

第一张图（左上）显示的是原始数据点，用不同颜色加以区分。算法首先找到方差最大的方向，将其标记为“成分 1”（Component 1）。这是数据中包含最多信息的方向（或向量），换句话说，沿着这个方向的特征之间最为相关。然后，算法找到与第一个方向正交（成直角）且包含最多信息的方向。在二维空间中，只有一个成直角的方向，但在更高维的空间中会有（无穷）多的正交方向。虽然这两个成分都画成箭头，但其头尾的位置并不重要。我们也可以将第一个成分画成从中心指向左上，而不是指向右下。利用这一过程找到的方向被称为**主成分**（principal component），因为它们是数据方差的主要方向。一般来说，主成分的个数与原始特征相同。

第二张图（右上）显示的是同样的数据，但现在将其旋转，使得第一主成分与 x 轴平行且第二主成分与 y 轴平行。在旋转之前，从数据中减去平均值，使得变换后的数据以零为中心。在 PCA 找到的旋转表示中，两个坐标轴是不相关的，也就是说，对于这种数据表示，除了对角线，相关矩阵全部为零。

我们可以通过仅保留一部分主成分来使用 PCA 进行降维。在这个例子中，我们可以仅保留第一个主成分，正如图 3-3 中第三张图所示（左下）。这将数据从二维数据集降为一维数据集。但要注意，我们没有保留原始特征之一，而是找到了最有趣的方向（第一张图中从左上到右下）并保留这一方向，即第一主成分。

最后，我们可以反向旋转并将平均值重新加到数据中。这样会得到图 3-3 最后一张图中的数据。这些数据点位于原始特征空间中，但我们仅保留了第一主成分中包含的信息。这种变换有时用于去除数据中的噪声影响，或者将主成分中保留的那部分信息可视化。

1. 将 PCA 应用于 `cancer` 数据集并可视化

PCA 最常见的应用之一就是将高维数据集可视化。正如第 1 章中所说，对于有两个以上特征的数据，很难绘制散点图。对于 Iris（鸢尾花）数据集，我们可以创建散点图矩阵（见第 1 章图 1-3），通过展示特征所有可能的两两组合来展示数据的局部图像。但如果我们想要查看乳腺癌数据集，即便用散点图矩阵也很困难。这个数据集包含 30 个特征，这就导致需要绘制 30 * 14 = 420 张散点图！我们永远不可能仔细观察所有这些图像，更不用说试图理解它们了。

不过我们可以使用一种更简单的可视化方法——对每个特征分别计算两个类别（良性肿瘤和恶性肿瘤）的直方图（见图 3-4）。

In[14]:

```
fig, axes = plt.subplots(15, 2, figsize=(10, 20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]

ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:, i], bins=50)
    ax[i].hist(malignant[:, i], bins=bins, color=mglearn.cm3(0), alpha=.5)
    ax[i].hist(benign[:, i], bins=bins, color=mglearn.cm3(2), alpha=.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Feature magnitude")
ax[0].set_ylabel("Frequency")
ax[0].legend(["malignant", "benign"], loc="best")
fig.tight_layout()
```

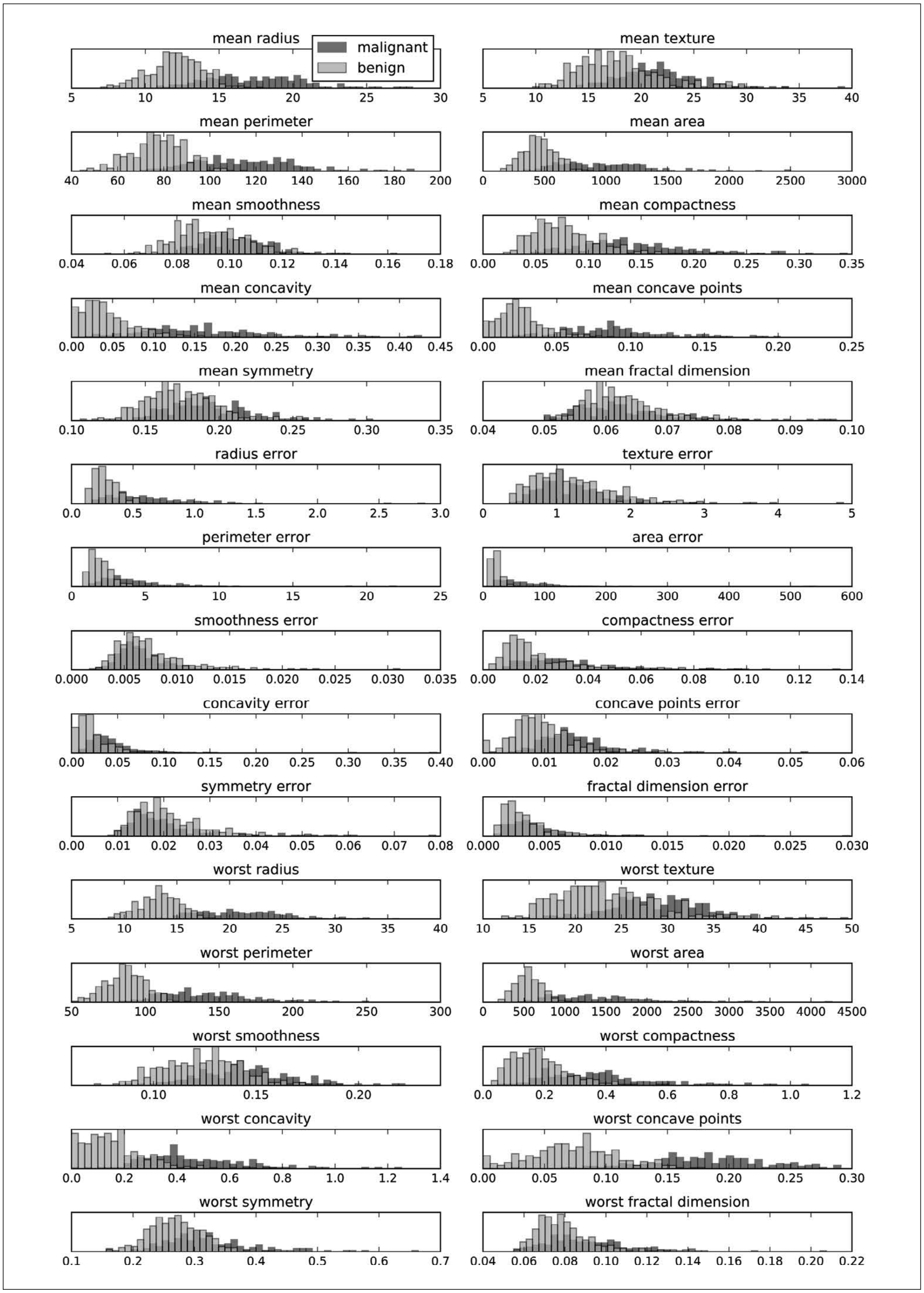


图 3-4：乳腺癌数据集中每个类别的特征直方图

这里我们为每个特征创建一个直方图，计算具有某一特征的数据点在特定范围内（叫作 bin）的出现频率。每张图都包含两个直方图，一个是良性类别的所有点（蓝色），一个是恶性类别的所有点（红色）。这样我们可以了解每个特征在两个类别中的分布情况，也可以猜测哪些特征能够更好地区分良性样本和恶性样本。例如，“smoothness error”特征似乎没有什么信息量，因为两个直方图大部分都重叠在一起，而“worst concave points”特征看起来信息量相当大，因为两个直方图的交集很小。

但是，这种图无法向我们展示变量之间的相互作用以及这种相互作用与类别之间的关系。利用 PCA，我们可以获取到主要的相互作用，并得到稍为完整的图像。我们可以找到前两个主成分，并在这个新的二维空间中用散点图将数据可视化。

在应用 PCA 之前，我们利用 `StandardScaler` 缩放数据，使每个特征的方差均为 1：

In[15]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()

scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

学习并应用 PCA 变换与应用预处理变换一样简单。我们将 PCA 对象实例化，调用 `fit` 方法找到主成分，然后调用 `transform` 来旋转并降维。默认情况下，`PCA` 仅旋转（并移动）数据，但保留所有的主成分。为了降低数据的维度，我们需要在创建 `PCA` 对象时指定想要保留的主成分个数：

In[16]:

```
from sklearn.decomposition import PCA
# 保留数据的前两个主成分
pca = PCA(n_components=2)
# 对乳腺癌数据拟合PCA模型
pca.fit(X_scaled)

# 将数据变换到前两个主成分的方向上
X_pca = pca.transform(X_scaled)
print("Original shape: {}".format(str(X_scaled.shape)))
print("Reduced shape: {}".format(str(X_pca.shape)))
```

Out[16]:

```
Original shape: (569, 30)
Reduced shape: (569, 2)
```

现在我们可以对前两个主成分作图（图 3-5）：

In[17]:

```
# 对第一个和第二个主成分作图，按类别着色
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca[:, 0], X_pca[:, 1], cancer.target)
plt.legend(cancer.target_names, loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

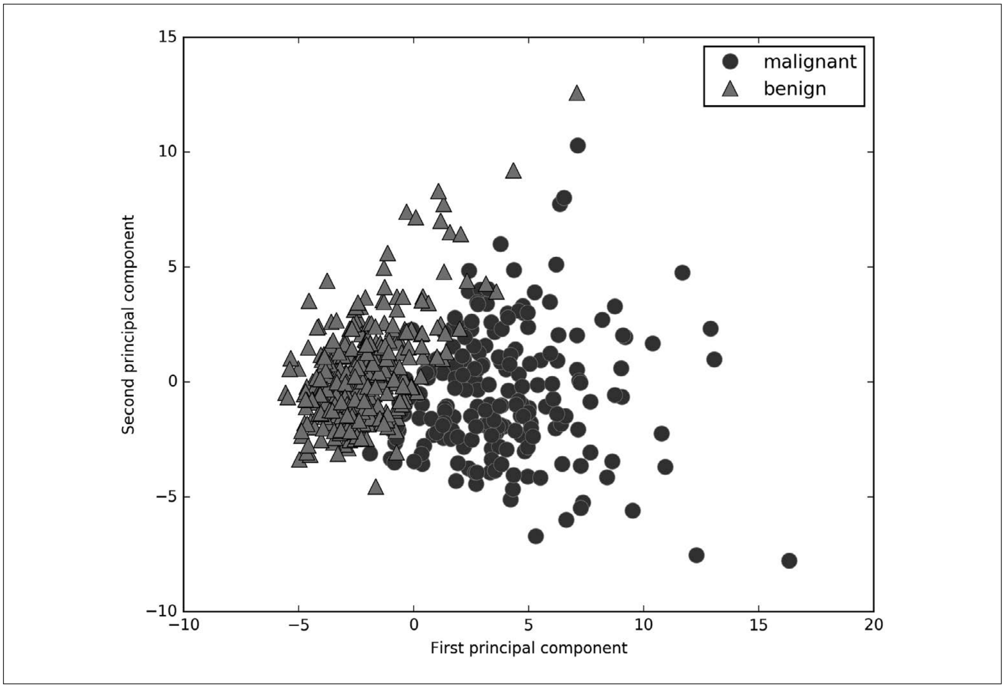


图 3-5：利用前两个主成分绘制乳腺癌数据集的二维散点图

重要的是要注意，PCA 是一种无监督方法，在寻找旋转方向时没有用到任何类别信息。它只是观察数据中的相关性。对于这里所示的散点图，我们绘制了第一主成分与第二主成分的关系，然后利用类别信息对数据点进行着色。你可以看到，在这个二维空间中两个类别被很好地分离。这让我们相信，即使是线性分类器（在这个空间中学习一条直线）也可以在区分这个两个类别时表现得相当不错。我们还可以看到，恶性点比良性点更加分散，这一点也可以在图 3-4 的直方图中看出来。

PCA 的一个缺点在于，通常不容易对图中的两个轴做出解释。主成分对应于原始数据中的方向，所以它们是原始特征的组合。但这些组合往往非常复杂，这一点我们很快就会看到。在拟合过程中，主成分被保存在 PCA 对象的 components_ 属性中：

In[18]:

```
print("PCA component shape: {}".format(pca.components_.shape))
```

Out[18]:

```
PCA component shape: (2, 30)
```

components_ 中的每一行对应于一个主成分，它们按重要性排序（第一主成分排在首位，以此类推）。列对应于 PCA 的原始特征属性，在本例中即为“mean radius”“mean texture”等。我们来看一下 components_ 的内容：

In[19]:

```
print("PCA components:\n{}".format(pca.components_))
```

Out[19]:

```
PCA components:
[[ 0.219  0.104  0.228  0.221  0.143  0.239  0.258  0.261  0.138  0.064
   0.206  0.017  0.211  0.203  0.015  0.17   0.154  0.183  0.042  0.103
   0.228  0.104  0.237  0.225  0.128  0.21   0.229  0.251  0.123  0.132]
[-0.234 -0.06  -0.215 -0.231  0.186  0.152  0.06  -0.035  0.19   0.367
 -0.106  0.09  -0.089 -0.152  0.204  0.233  0.197  0.13   0.184  0.28
 -0.22  -0.045 -0.2   -0.219  0.172  0.144  0.098 -0.008  0.142  0.275]]
```

我们还可以用热图将系数可视化（图 3-6），这可能更容易理解：

In[20]:

```
plt.matshow(pca.components_, cmap='viridis')
plt.yticks([0, 1], ["First component", "Second component"])
plt.colorbar()
plt.xticks(range(len(cancer.feature_names)),
           cancer.feature_names, rotation=60, ha='left')
plt.xlabel("Feature")
plt.ylabel("Principal components")
```

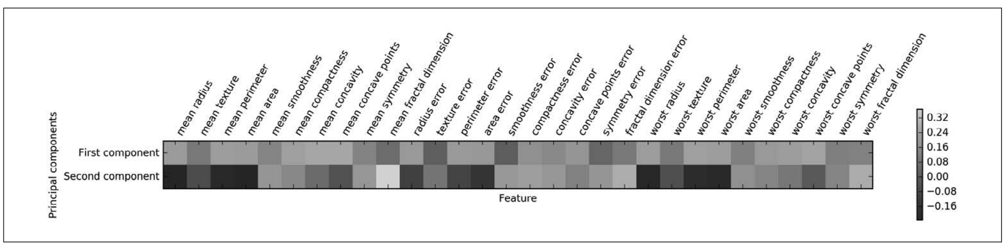


图 3-6：乳腺癌数据集前两个主成分的热图

你可以看到，在第一个主成分中，所有特征的符号相同（均为正，但前面我们提到过，箭头指向哪个方向无关紧要）。这意味着在所有特征之间存在普遍的相关性。如果一个测量值较大的话，其他的测量值可能也较大。第二个主成分的符号有正有负，而且两个主成分都包含所有 30 个特征。这种所有特征的混合使得解释图 3-6 中的坐标轴变得十分困难。

2. 特征提取的特征脸

前面提到过，PCA 的另一个应用是特征提取。特征提取背后的思想是，可以找到一种数据表示，比给定的原始表示更适合于分析。特征提取很有用，它的一个很好的应用实例就是图像。图像由像素组成，通常存储为红绿蓝（RGB）强度。图像中的对象通常由上千个像素组成，它们只有放在一起才有意义。

我们将给出用 PCA 对图像做特征提取的一个简单应用，即处理 Wild 数据集 Labeled Faces（标记人脸）中的人脸图像。这一数据集包含从互联网下载的名人脸部图像，它包含从 21 世纪初开始的政治家、歌手、演员和运动员的人脸图像。我们使用这些图像的灰度版本，并将它们按比例缩小以加快处理速度。你可以在图 3-7 中看到其中一些图像：

In[21]:

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
```

```
image_shape = people.images[0].shape

fix, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
    ax.imshow(image)
    ax.set_title(people.target_names[target])
```



图 3-7：来自 Wild 数据集中 Labeled Faces 的一些图像

一共有 3023 张图像，每张大小为 87 像素 ×65 像素，分别属于 62 个不同的人：

In[22]:

```
print("people.images.shape: {}".format(people.images.shape))
print("Number of classes: {}".format(len(people.target_names)))
```

Out[22]:

```
people.images.shape: (3023, 87, 65)
Number of classes: 62
```

但这个数据集有些偏斜，其中包含 George W. Bush（小布什）和 Colin Powell（科林·鲍威尔）的大量图像，正如你在下面所见：

In[23]:

```
# 计算每个目标出现的次数
counts = np.bincount(people.target)
# 将次数与目标名称一起打印出来
for i, (count, name) in enumerate(zip(counts, people.target_names)):
    print("{0:25} {1:3}".format(name, count), end='   ')
    if (i + 1) % 3 == 0:
        print()
```

Out[23]:

```
Alejandro Toledo           39     Alvaro Uribe               35
Amelie Mauresmo            21     Andre Agassi               36
Angelina Jolie             20     Ariel Sharon               77
Arnold Schwarzenegger      42     Atal Bihari Vajpayee       24
Bill Clinton               29     Carlos Menem               21
Colin Powell              236     David Beckham              31
Donald Rumsfeld           121     George Robertson           22
George W Bush             530     Gerhard Schroeder         109
Gloria Macapagal Arroyo    44     Gray Davis                 26
Guillermo Coria            30     Hamid Karzai               22
Hans Blix                  39     Hugo Chavez                71
Igor Ivanov                20     Jack Straw                 28
Jacques Chirac             52     Jean Chretien              55
Jennifer Aniston           21     Jennifer Capriati          42
Jennifer Lopez             21     Jeremy Greenstock          24
Jiang Zemin                20     John Ashcroft              53
John Negroponte            31     Jose Maria Aznar           23
Juan Carlos Ferrero        28     Junichiro Koizumi          60
Kofi Annan                 32     Laura Bush                 41
Lindsay Davenport          22     Lleyton Hewitt             41
Luiz Inacio Lula da Silva  48     Mahmoud Abbas              29
Megawati Sukarnoputri      33     Michael Bloomberg          20
Naomi Watts                22     Nestor Kirchner            37
Paul Bremer                20     Pete Sampras               22
Recep Tayyip Erdogan       30     Ricardo Lagos              27
Roh Moo-hyun               32     Rudolph Giuliani           26
Saddam Hussein             23     Serena Williams            52
Silvio Berlusconi          33     Tiger Woods                23
Tom Daschle                25     Tom Ridge                  33
Tony Blair                144     Vicente Fox                32
Vladimir Putin             49     Winona Ryder               24
```

为了降低数据偏斜，我们对每个人最多只取 50 张图像（否则，特征提取将会被 George W. Bush 的可能性大大影响）：

In[24]:

```
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1

X_people = people.data[mask]
y_people = people.target[mask]

# 将灰度值缩放到0到1之间，而不是在0到255之间
# 以得到更好的数据稳定性
X_people = X_people / 255.
```

人脸识别的一个常见任务就是看某个前所未见的人脸是否属于数据库中的某个已知人物。这在照片收集、社交媒体和安全应用中都有应用。解决这个问题的方法之一就是构建一个分类器，每个人都是一个单独的类别。但人脸数据库中通常有许多不同的人，而同一个人的图像很少（也就是说，每个类别的训练样例很少）。这使得大多数分类器的训练都很困难。另外，通常你还想要能够轻松添加新的人物，不需要重新训练一个大型模型。

一种简单的解决方法是使用单一最近邻分类器，寻找与你要分类的人脸最为相似的人脸。这个分类器原则上可以处理每个类别只有一个训练样例的情况。下面看一下 KNeighborsClassifier 的表现如何：

In[25]:

```
from sklearn.neighbors import KNeighborsClassifier
# 将数据分为训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
# 使用一个邻居构建KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Test set score of 1-nn: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[25]:

```
Test set score of 1-nn: 0.27
```

我们得到的精度为 26.6%。对于包含 62 个类别的分类问题来说，这实际上不算太差（随机猜测的精度约为 1/62=1.5%），但也不算好。我们每识别四次仅正确识别了一个人。

这里就可以用到 PCA。想要度量人脸的相似度，计算原始像素空间中的距离是一种相当糟糕的方法。用像素表示来比较两张图像时，我们比较的是每个像素的灰度值与另一张图像对应位置的像素灰度值。这种表示与人们对人脸图像的解释方式有很大不同，使用这种原始表示很难获取到面部特征。例如，如果使用像素距离，那么将人脸向右移动一个像素将会发生巨大的变化，得到一个完全不同的表示。我们希望，使用沿着主成分方向的距离可以提高精度。这里我们启用 PCA 的**白化**（whitening）选项，它将主成分缩放到相同的尺度。变换后的结果与使用 StandardScaler 相同。再次使用图 3-3 中的数据，白化不仅对应于旋转数据，还对应于缩放数据使其形状是圆形而不是椭圆（参见图 3-8）：

In[26]:

```
mglearn.plots.plot_pca_whitening()
```

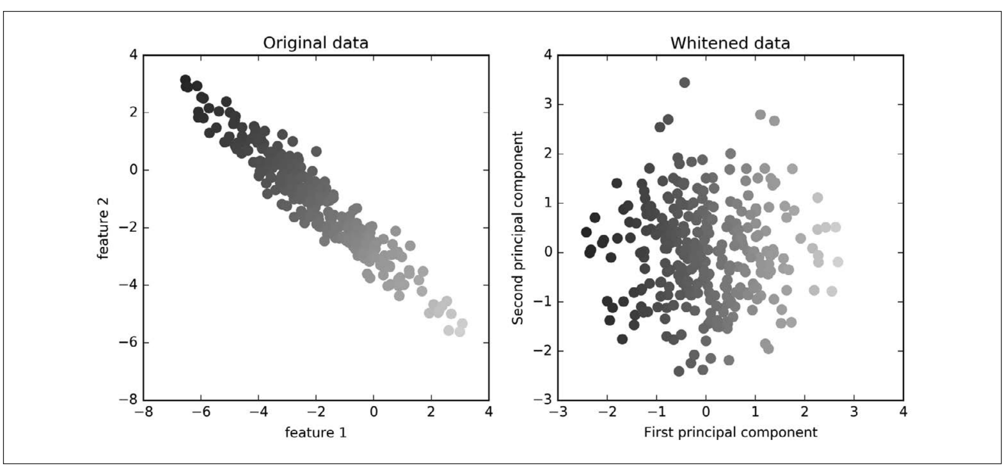


图 3-8：利用启用白化的 PCA 进行数据变换

我们对训练数据拟合 PCA 对象，并提取前 100 个主成分。然后对训练数据和测试数据进行变换：

In[27]:

```
pca = PCA(n_components=100, whiten=True, random_state=0).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

print("X_train_pca.shape: {}".format(X_train_pca.shape))
```

Out[27]:

```
X_train_pca.shape: (1547, 100)
```

新数据有 100 个特征，即前 100 个主成分。现在，可以对新表示使用单一最近邻分类器来将我们的图像分类：

In[28]:

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("Test set accuracy: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

Out[28]:

```
Test set accuracy: 0.36
```

我们的精度有了相当显著的提高，从 26.6% 提升到 35.7%，这证实了我们的直觉，即主成分可能提供了一种更好的数据表示。

对于图像数据，我们还可以很容易地将找到的主成分可视化。请记住，成分对应于输入空间里的方向。这里的输入空间是 87 像素 ×65 像素的灰度图像，所以在这个空间中的方向也是 87 像素 ×65 像素的灰度图像。

我们来看一下前几个主成分（图 3-9）：

In[29]:

```
print("pca.components_.shape: {}".format(pca.components_.shape))
```

Out[29]:

```
pca.components_.shape: (100, 5655)
```

In[30]:

```
fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(pca.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape),
              cmap='viridis')
    ax.set_title("{}. component".format((i + 1)))
```

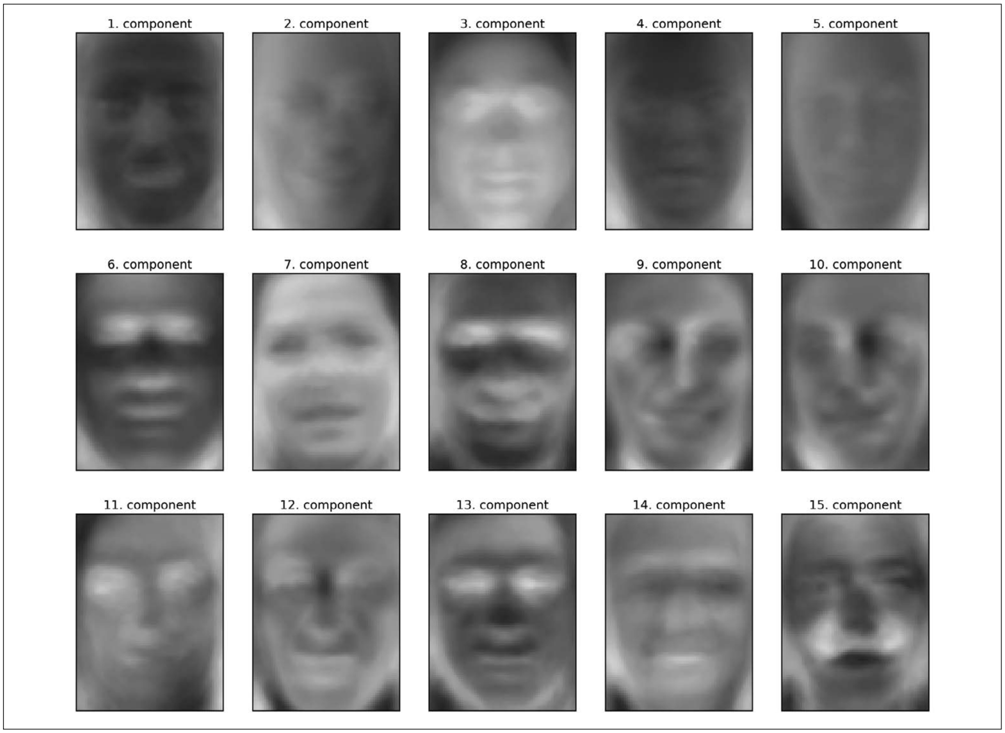


图 3-9：人脸数据集前 15 个主成分的成分向量

虽然我们肯定无法理解这些成分的所有内容，但可以猜测一些主成分捕捉到了人脸图像的哪些方面。第一个主成分似乎主要编码的是人脸与背景的对比，第二个主成分编码的是人脸左半部分和右半部分的明暗程度差异，如此等等。虽然这种表示比原始像素值的语义稍强，但它仍与人们感知人脸的方式相去甚远。由于 PCA 模型是基于像素的，因此人脸的相对位置（眼睛、下巴和鼻子的位置）和明暗程度都对两张图像在像素表示中的相似程度有很大影响。但人脸的相对位置和明暗程度可能并不是人们首先感知的内容。在要求人们评价人脸的相似度时，他们更可能会使用年龄、性别、面部表情和发型等属性，而这些属性很难从像素强度中推断出来。重要的是要记住，算法对数据（特别是视觉数据，比如人们非常熟悉的图像）的解释通常与人类的解释方式大不相同。

不过让我们回到 PCA 的具体案例。我们对 PCA 变换的介绍是：先旋转数据，然后删除方差较小的成分。另一种有用的解释是尝试找到一些数字（PCA 旋转后的新特征值），使我们可以将测试点表示为主成分的加权求和（见图 3-10）。

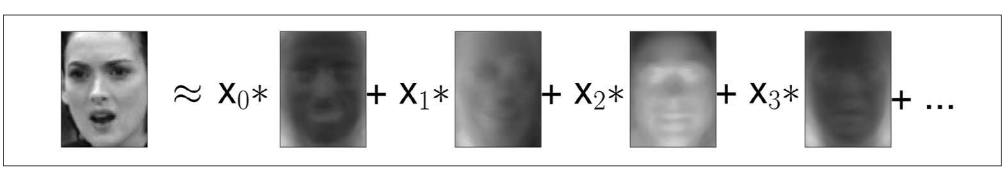


图 3-10：图解 PCA：将图像分解为成分的加权求和

这里 x_0、x_1 等是这个数据点的主成分的系数，换句话说，它们是图像在旋转后的空间中的表示。

我们还可以用另一种方法来理解 PCA 模型，就是仅使用一些成分对原始数据进行重建。在图 3-3 中，在去掉第二个成分并来到第三张图之后，我们反向旋转并重新加上平均值，这样就在原始空间中获得去掉第二个成分的新数据点，正如最后一张图所示。我们可以对人脸做类似的变换，将数据降维到只包含一些主成分，然后反向旋转回到原始空间。回到原始特征空间可以通过 `inverse_transform` 方法来实现。这里我们分别利用 10 个、50 个、100 个和 500 个成分对一些人脸进行重建并将其可视化（图 3-11）：

In[32]:

```
mglearn.plots.plot_pca_faces(X_train, X_test, image_shape)
```



图 3-11：利用越来越多的主成分对三张人脸图像进行重建

可以看到，在仅使用前 10 个主成分时，仅捕捉到了图片的基本特点，比如人脸方向和明暗程度。随着使用的主成分越来越多，图像中也保留了越来越多的细节。这对应于图 3-10 的求和中包含越来越多的项。如果使用的成分个数与像素个数相等，意味着我们在旋转后不会丢弃任何信息，可以完美重建图像。

我们还可以尝试使用 PCA 的前两个主成分，将数据集中的所有人脸在散点图中可视化（图 3-12），其类别在图中给出。这与我们对 `cancer` 数据集所做的类似：

In[33]:

```
mglearn.discrete_scatter(X_train_pca[:, 0], X_train_pca[:, 1], y_train)
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

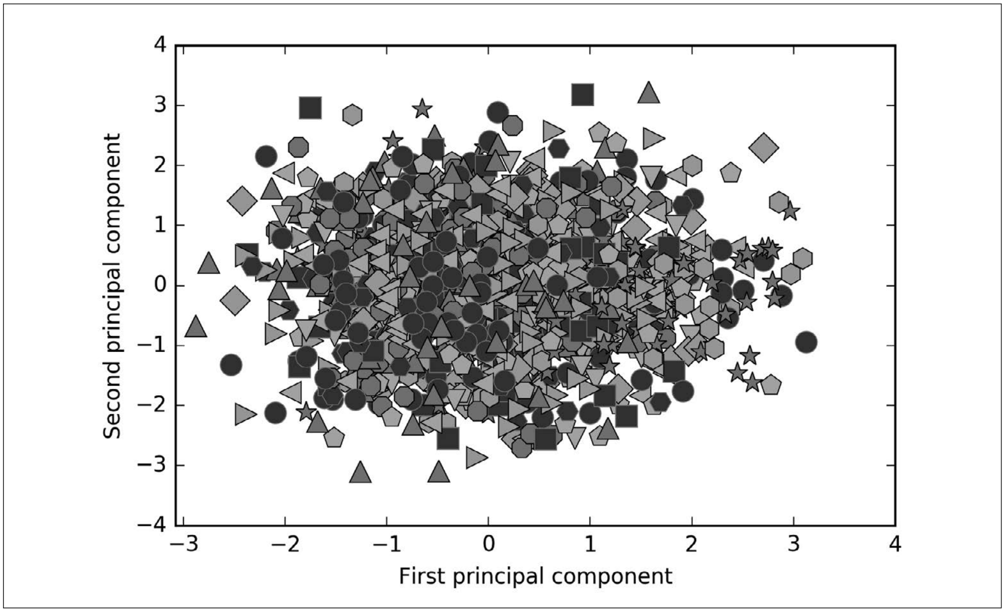


图 3-12：利用前两个主成分绘制人脸数据集的散点图（cancer 数据集的对应图像见图 3-5）

如你所见，如果我们只使用前两个主成分，整个数据只是一大团，看不到类别之间的分界。这并不意外，因为即使有 10 个成分（正如图 3-11 所示），PCA 也仅捕捉到人脸非常粗略的特征。

3.4.2　非负矩阵分解

非负矩阵分解（non-negative matrix factorization，NMF）是另一种无监督学习算法，其目的在于提取有用的特征。它的工作原理类似于 PCA，也可以用于降维。与 PCA 相同，我们试图将每个数据点写成一些分量的加权求和，正如图 3-10 所示。但在 PCA 中，我们想要的是正交分量，并且能够解释尽可能多的数据方差；而在 NMF 中，我们希望分量和系数均为非负，也就是说，我们希望分量和系数都大于或等于 0。因此，这种方法只能应用于每个特征都是非负的数据，因为非负分量的非负求和不可能变为负值。

将数据分解成非负加权求和的这个过程，对由多个独立源相加（或叠加）创建而成的数据特别有用，比如多人说话的音轨或包含多种乐器的音乐。在这种情况下，NMF 可以识别出组成合成数据的原始分量。总的来说，与 PCA 相比，NMF 得到的分量更容易解释，因为负的分量和系数可能会导致难以解释的抵消效应（cancellation effect）。举个例子，图 3-9 中的特征脸同时包含正数和负数，我们在 PCA 的说明中也提到过，正负号实际上是任意的。在将 NMF 应用于人脸数据集之前，我们先来简要回顾一下模拟数据。

1. 将 NMF 应用于模拟数据

与使用 PCA 不同，我们需要保证数据是正的，NMF 能够对数据进行操作。这说明数据相对于原点 (0, 0) 的位置实际上对 NMF 很重要。因此，你可以将提取出来的非负分量看作是从 (0, 0) 到数据的方向。

下面的例子（图 3-13）给出了 NMF 在二维玩具数据上的结果：

In[34]:

```
mglearn.plots.plot_nmf_illustration()
```

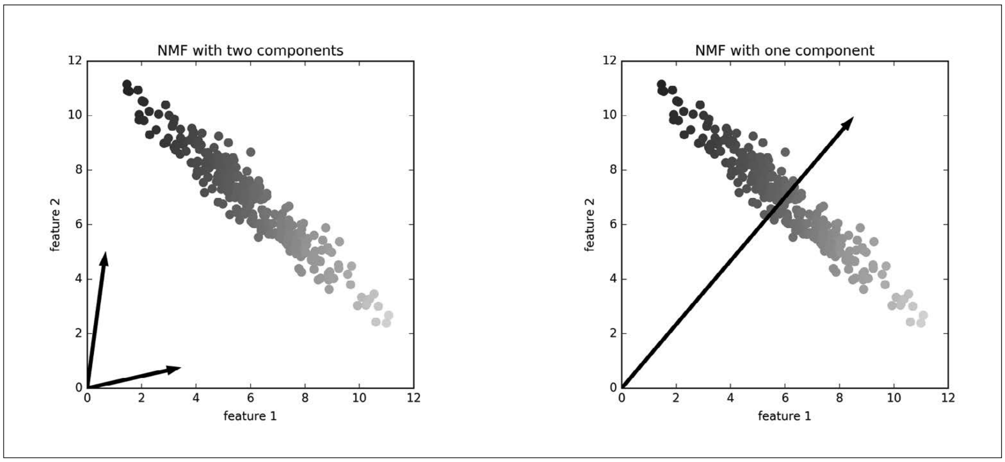


图 3-13：两个分量的非负矩阵分解（左）和一个分量的非负矩阵分解（右）找到的分量

对于两个分量的 NMF（如左图所示），显然所有数据点都可以写成这两个分量的正数组合。如果有足够多的分量能够完美地重建数据（分量个数与特征个数相同），那么算法会选择指向数据极值的方向。

如果我们仅使用一个分量，那么 NMF 会创建一个指向平均值的分量，因为指向这里可以对数据做出最好的解释。你可以看到，与 PCA 不同，减少分量个数不仅会删除一些方向，而且会创建一组完全不同的分量！ NMF 的分量也没有按任何特定方法排序，所以不存在"第一非负分量"：所有分量的地位平等。

NMF 使用了随机初始化，根据随机种子的不同可能会产生不同的结果。在相对简单的情况下（比如两个分量的模拟数据），所有数据都可以被完美地解释，那么随机性的影响很小（虽然可能会影响分量的顺序或尺度）。在更加复杂的情况下，影响可能会很大。

2. 将 NMF 应用于人脸图像

现在我们将 NMF 应用于之前用过的 Wild 数据集中的 Labeled Faces。NMF 的主要参数是我们想要提取的分量个数。通常来说，这个数字要小于输入特征的个数（否则的话，将每个像素作为单独的分量就可以对数据进行解释）。

首先，我们来观察分量个数如何影响 NMF 重建数据的好坏（图 3-14）：

In[35]:

```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```



图 3-14：利用越来越多分量的 NMF 重建三张人脸图像

反向变换的数据质量与使用 PCA 时类似，但要稍差一些。这是符合预期的，因为 PCA 找到的是重建的最佳方向。NMF 通常并不用于对数据进行重建或编码，而是用于在数据中寻找有趣的模式。

我们尝试仅提取一部分分量（比如 15 个），初步观察一下数据。其结果见图 3-15。

In[36]:

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)

fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("{}. component".format(i))
```



图 3-15：使用 15 个分量的 NMF 在人脸数据集上找到的分量

这些分量都是正的，因此比图 3-9 所示的 PCA 分量更像人脸原型。例如，你可以清楚地看到，分量 3（component 3）显示了稍微向右转动的人脸，而分量 7（component 7）则显示了稍微向左转动的人脸。我们来看一下这两个分量特别大的那些图像，分别如图 3-16 和图 3-17 所示。

In[37]:

```
compn = 3
# 按第3个分量排序，绘制前10张图像
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))

compn = 7
# 按第7个分量排序，绘制前10张图像
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
```



图 3-16：分量 3 系数较大的人脸



图 3-17：分量 7 系数较大的人脸

正如所料，分量 3 系数较大的人脸都是向右看的人脸（图 3-16），而分量 7 系数较大的人脸都向左看（图 3-17）。如前所述，提取这样的模式最适合于具有叠加结构的数据，包括音频、基因表达和文本数据。我们通过一个模拟数据的例子来看一下这种用法。

假设我们对一个信号感兴趣，它是三个不同信号源合成的（图 3-18）：

In[38]:

```
S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
```

```
plt.plot(S, '-')
plt.xlabel("Time")
plt.ylabel("Signal")
```

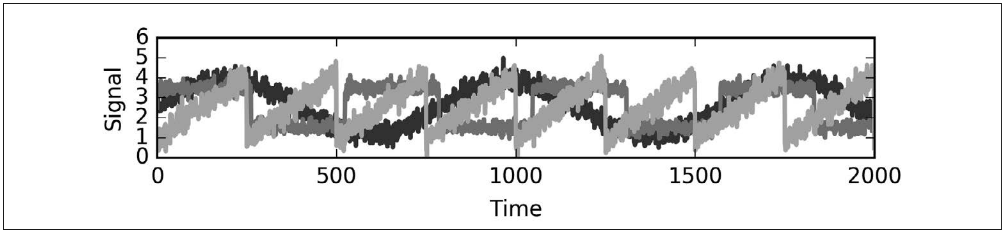


图 3-18：原始信号源

不幸的是，我们无法观测到原始信号，只能观测到三个信号的叠加混合。我们想要将混合信号分解为原始分量。假设我们有许多种不同的方法来观测混合信号（比如有 100 台测量装置），每种方法都为我们提供了一系列测量结果。

In[39]:

```
# 将数据混合成100维的状态
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Shape of measurements: {}".format(X.shape))
```

Out[39]:

```
Shape of measurements: (2000, 100)
```

我们可以用 NMF 来还原这三个信号：

In[40]:

```
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Recovered signal shape: {}".format(S_.shape))
```

Out[40]:

```
Recovered signal shape: (2000, 3)
```

为了对比，我们也应用了 PCA：

In[41]:

```
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

图 3-19 给出了 NMF 和 PCA 发现的信号活动：

In[42]:

```
models = [X, S, S_, H]
names = ['Observations (first three measurements)',
         'True sources',
         'NMF recovered signals',
         'PCA recovered signals']
```

```
fig, axes = plt.subplots(4, figsize=(8, 4), gridspec_kw={'hspace': .5},
                         subplot_kw={'xticks': (), 'yticks': ()})

for model, name, ax in zip(models, names, axes):
    ax.set_title(name)
    ax.plot(model[:, :3], '-')
```

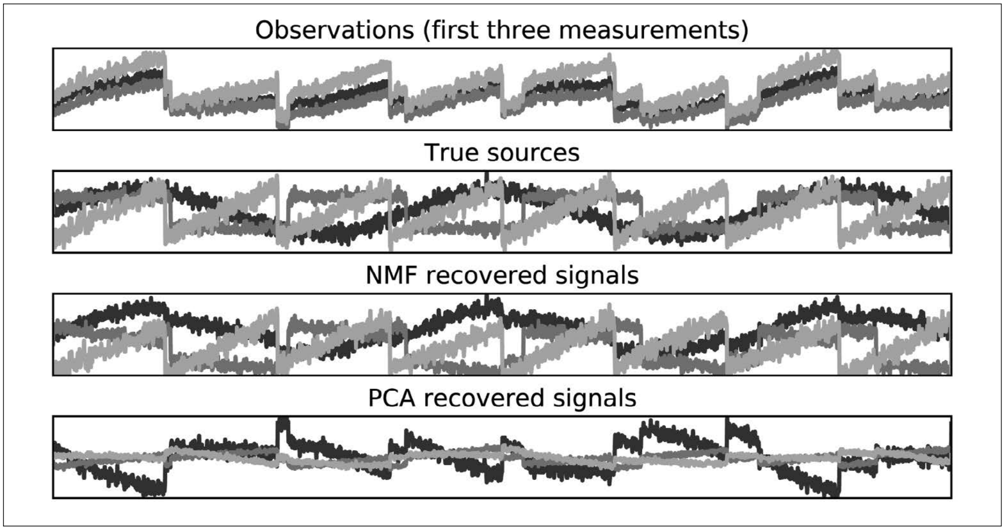


图 3-19：利用 NMF 和 PCA 还原混合信号源

图中包含来自 X 的 100 次测量中的 3 次，用于参考。可以看到，NMF 在发现原始信号源时得到了不错的结果，而 PCA 则失败了，仅使用第一个成分来解释数据中的大部分变化。要记住，NMF 生成的分量是没有顺序的。在这个例子中，NMF 分量的顺序与原始信号完全相同（参见三条曲线的颜色），但这纯属偶然。

还有许多其他算法可用于将每个数据点分解为一系列固定分量的加权求和，正如 PCA 和 NMF 所做的那样。讨论所有这些算法已超出了本书的范围，而且描述对分量和系数的约束通常要涉及概率论。如果你对这种类型的模式提取感兴趣，我们推荐你学习 scikit-learn 用户指南中关于独立成分分析（ICA）、因子分析（FA）和稀疏编码（字典学习）等的内容，所有这些内容都可以在关于分解方法的页面中找到（http://scikit-learn.org/stable/modules/decomposition.html）。

3.4.3　用t-SNE进行流形学习

虽然 PCA 通常是用于变换数据的首选方法，使你能够用散点图将其可视化，但这一方法的性质（先旋转然后减少方向）限制了其有效性，正如我们在 Wild 数据集 Labeled Faces 的散点图中所看到的那样。有一类用于可视化的算法叫作**流形学习算法**（manifold learning algorithm），它允许进行更复杂的映射，通常也可以给出更好的可视化。其中特别有用的一个就是 t-SNE 算法。

流形学习算法主要用于可视化，因此很少用来生成两个以上的新特征。其中一些算法（包括 t-SNE）计算训练数据的一种新表示，但不允许变换新数据。这意味着这些算法不能用于测试集：更确切地说，它们只能变换用于训练的数据。流形学习对探索性数据分析是很有用的，但如果最终目标是监督学习的话，则很少使用。t-SNE 背后的思想是找到数据的一个二维表示，尽可能地保持数据点之间的距离。t-SNE 首先给出每个数据点的随机二维表示，然后尝试让在原始特征空间中距离较近的点更加靠近，原始特征空间中相距较远的点更加远离。t-SNE 重点关注距离较近的点，而不是保持距离较远的点之间的距离。换句话说，它试图保存那些表示哪些点比较靠近的信息。

我们将对 `scikit-learn` 包含的一个手写数字数据集[2]应用 t-SNE 流形学习算法。在这个数据集中，每个数据点都是 0 到 9 之间手写数字的一张 8×8 灰度图像。图 3-20 给出了每个类别的一个例子。

In[43]:
```
from sklearn.datasets import load_digits
digits = load_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                         subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img)
```

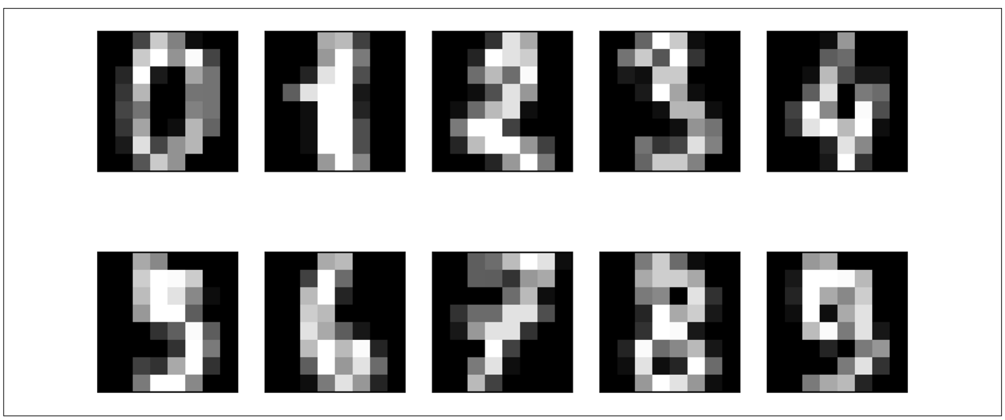


图 3-20：`digits` 数据集的示例图像

我们用 PCA 将降到二维的数据可视化。我们对前两个主成分作图，并按类别对数据点着色（图 3-21）：

In[44]:
```
# 构建一个PCA模型
pca = PCA(n_components=2)
pca.fit(digits.data)
# 将digits数据变换到前两个主成分的方向上
```

注 2：不要与更大的 MNIST 数据集弄混。

```
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
    # 将数据实际绘制成文本，而不是散点
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

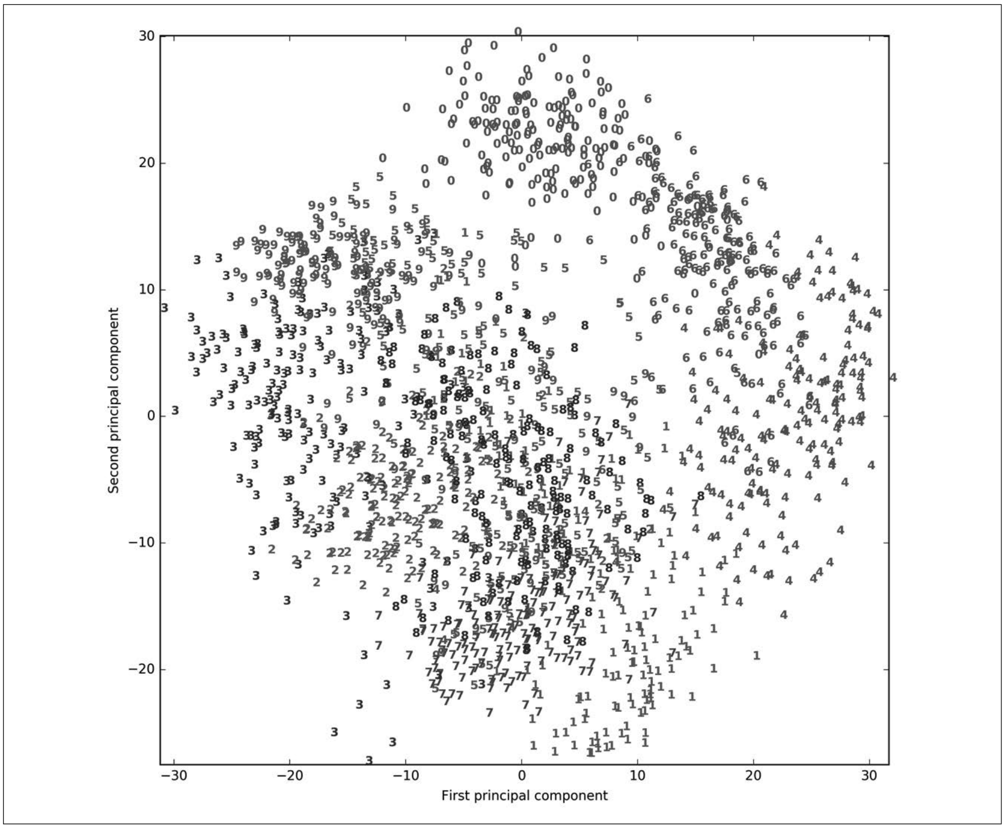


图 3-21：利用前两个主成分绘制 digits 数据集的散点图

实际上，这里我们用每个类别对应的数字作为符号来显示每个类别的位置。利用前两个主成分可以将数字 0、6 和 4 相对较好地分开，尽管仍有重叠。大部分其他数字都大量重叠在一起。

我们将 t-SNE 应用于同一个数据集，并对结果进行比较。由于 t-SNE 不支持变换新数据，所以 TSNE 类没有 transform 方法。我们可以调用 fit_transform 方法来代替，它会构建模

型并立刻返回变换后的数据（见图 3-22）：

In[45]:

```
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# 使用fit_transform而不是fit，因为TSNE没有transform方法
digits_tsne = tsne.fit_transform(digits.data)
```

In[46]:

```
plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # 将数据实际绘制成文本，而不是散点
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE feature 0")
plt.xlabel("t-SNE feature 1")
```

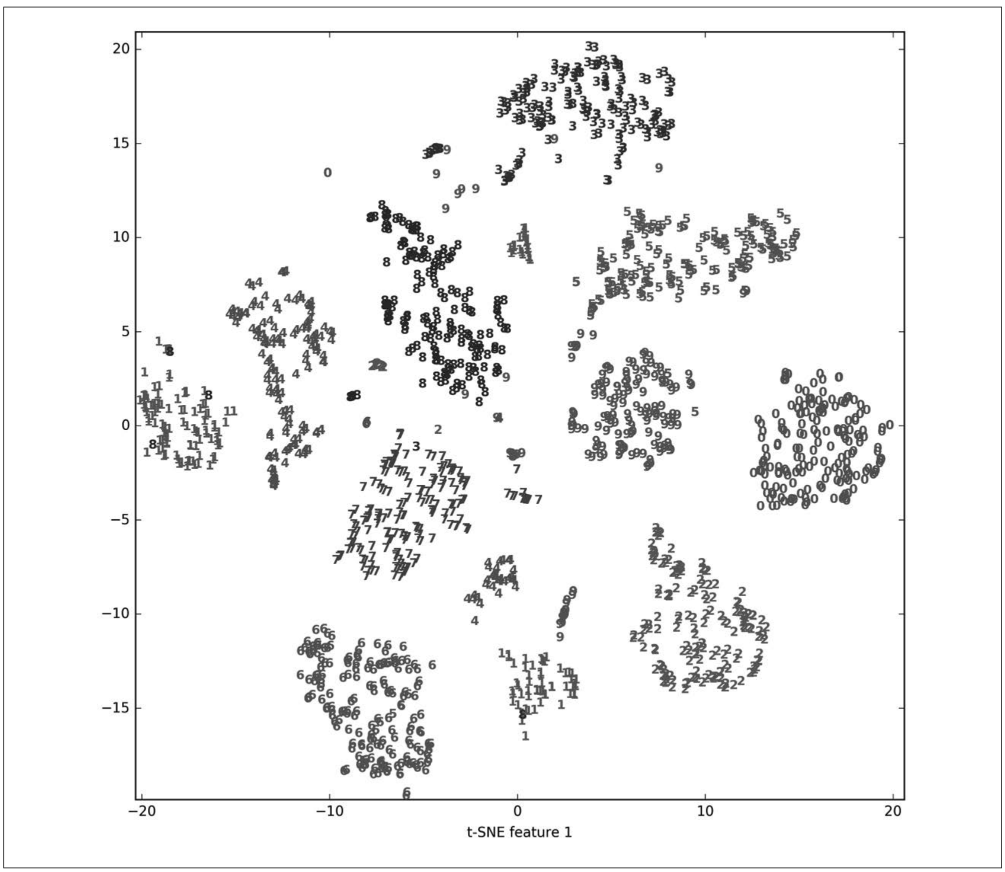


图 3-22：利用 t-SNE 找到的两个分量绘制 digits 数据集的散点图

t-SNE 的结果非常棒。所有类别都被明确分开。数字 1 和 9 被分成几块，但大多数类别都形成一个密集的组。要记住，这种方法并不知道类别标签：它完全是无监督的。但它能够找到数据的一种二维表示，仅根据原始空间中数据点之间的靠近程度就能够将各个类别明确分开。

t-SNE 算法有一些调节参数，虽然默认参数的效果通常就很好。你可以尝试修改 `perplexity` 和 `early_exaggeration`，但作用一般很小。

3.5 聚类

我们前面说过，**聚类**（clustering）是将数据集划分成组的任务，这些组叫作簇（cluster）。其目标是划分数据，使得一个簇内的数据点非常相似且不同簇内的数据点非常不同。与分类算法类似，聚类算法为每个数据点分配（或预测）一个数字，表示这个点属于哪个簇。

3.5.1 k均值聚类

k 均值聚类是最简单也最常用的聚类算法之一。它试图找到代表数据特定区域的**簇中心**（cluster center）。算法交替执行以下两个步骤：将每个数据点分配给最近的簇中心，然后将每个簇中心设置为所分配的所有数据点的平均值。如果簇的分配不再发生变化，那么算法结束。下面的例子（图 3-23）在一个模拟数据集上对这一算法进行说明：

In[47]:

```
mglearn.plots.plot_kmeans_algorithm()
```

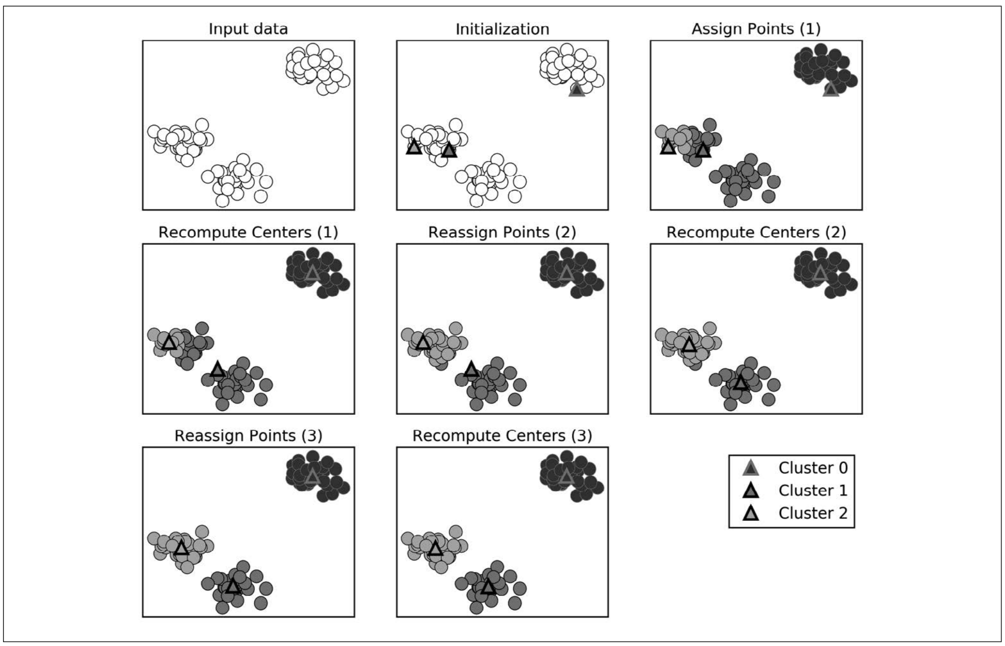


图 3-23：输入数据与 k 均值算法的三个步骤

簇中心用三角形表示，而数据点用圆形表示。颜色表示簇成员。我们指定要寻找三个簇，所以通过声明三个随机数据点为簇中心来将算法初始化（见图中“Initialization”/“初始化”）。然后开始迭代算法。首先，每个数据点被分配给距离最近的簇中心（见图中“Assign Points (1)”/“分配数据点（1）”）。接下来，将簇中心修改为所分配点的平均值（见图中“Recompute Centers (1)”/“重新计算中心（1）”）。然后将这一过程再重复两次。在第三次迭代之后，为簇中心分配的数据点保持不变，因此算法结束。

给定新的数据点，k 均值会将其分配给最近的簇中心。下一个例子（图 3-24）展示了图 3-23 学到的簇中心的边界：

In[48]:

```
mglearn.plots.plot_kmeans_boundaries()
```

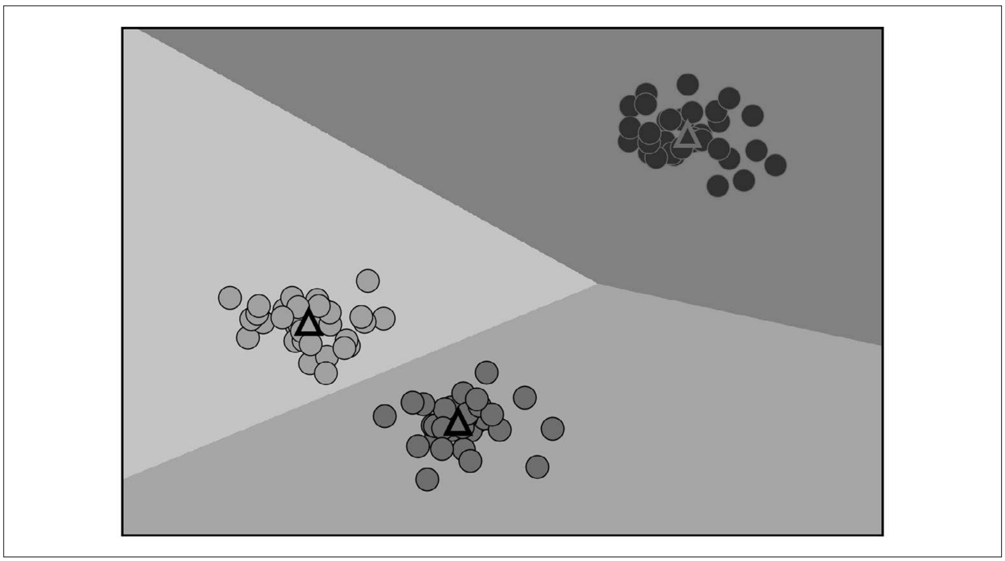


图 3-24：k 均值算法找到的簇中心和簇边界

用 `scikit-learn` 应用 k 均值相当简单。下面我们将其应用于上图中的模拟数据。我们将 `KMeans` 类实例化，并设置我们要寻找的簇个数[3]。然后对数据调用 `fit` 方法：

In[49]:

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# 生成模拟的二维数据
X, y = make_blobs(random_state=1)

# 构建聚类模型
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

注 3：如果不指定 `n_clusters`，它的默认值是 8。使用这个值并没有什么特别的原因。

算法运行期间，为 X 中的每个训练数据点分配一个簇标签。你可以在 kmeans.labels_ 属性中找到这些标签：

In[50]:
```
print("Cluster memberships:\n{}".format(kmeans.labels_))
```

Out[50]:
```
Cluster memberships:
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

因为我们要找的是 3 个簇，所以簇的编号是 0 到 2。

你也可以用 predict 方法为新数据点分配簇标签。预测时会将最近的簇中心分配给每个新数据点，但现有模型不会改变。对训练集运行 predict 会返回与 labels_ 相同的结果：

In[51]:
```
print(kmeans.predict(X))
```

Out[51]:
```
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

可以看到，聚类算法与分类算法有些相似，每个元素都有一个标签。但并不存在真实的标签，因此标签本身并没有**先验**意义。我们回到之前讨论过的人脸图像聚类的例子。聚类的结果可能是，算法找到的第 3 个簇仅包含你朋友 Bela 的面孔。但只有在查看图片之后才能知道这一点，而且数字 3 是任意的。算法给你的唯一信息就是所有标签为 3 的人脸都是相似的。

对于我们刚刚在二维玩具数据集上运行的聚类算法，这意味着我们不应该为其中一组的标签是 0、另一组的标签是 1 这一事实赋予任何意义。再次运行该算法可能会得到不同的簇编号，原因在于初始化的随机性质。

下面又给出了这个数据的图像（图 3-25）。簇中心被保存在 cluster_centers_ 属性中，我们用三角形表示它们：

In[52]:
```
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
```

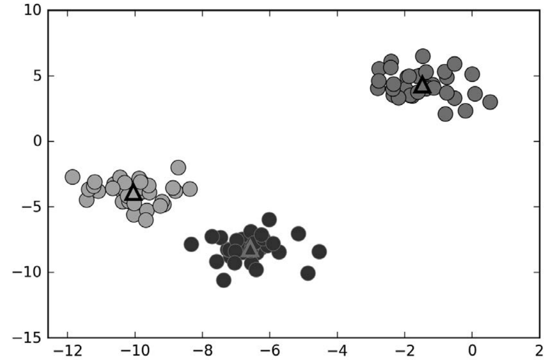


图 3-25：3 个簇的 k 均值算法找到的簇分配和簇中心

我们也可以使用更多或更少的簇中心（图 3-26）：

In[53]:

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# 使用2个簇中心：
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])

# 使用5个簇中心：
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])
```

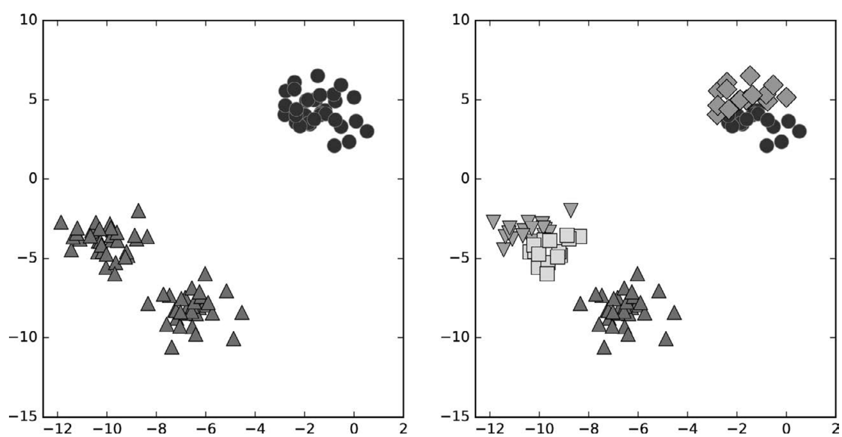


图 3-26：使用 2 个簇（左）和 5 个簇（右）的 k 均值算法找到的簇分配

1. k均值的失败案例

即使你知道给定数据集中簇的“正确”个数，k 均值可能也不是总能找到它们。每个簇仅由其中心定义，这意味着每个簇都是凸形（convex）。因此，k 均值只能找到相对简单的形状。k 均值还假设所有簇在某种程度上具有相同的“直径”，它总是将簇之间的边界刚好画在簇中心的中间位置。有时这会导致令人惊讶的结果，如图 3-27 所示：

In[54]:

```
X_varied, y_varied = make_blobs(n_samples=200,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)

mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["cluster 0", "cluster 1", "cluster 2"], loc='best')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

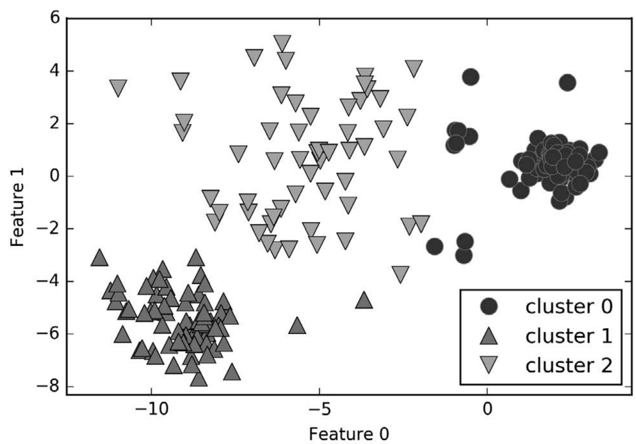


图 3-27：簇的密度不同时，k 均值找到的簇分配

你可能会认为，左下方的密集区域是第一个簇，右上方的密集区域是第二个，中间密度较小的区域是第三个。但事实上，簇 0 和簇 1 都包含一些远离簇中其他点的点。

k 均值还假设所有方向对每个簇都同等重要。图 3-28 显示了一个二维数据集，数据中包含明确分开的三部分。但是这三部分被沿着对角线方向拉长。由于 k 均值仅考虑到最近簇中心的距离，所以它无法处理这种类型的数据：

In[55]:

```
# 生成一些随机分组数据
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

# 变换数据使其拉长
transformation = rng.normal(size=(2, 2))
```

```
X = np.dot(X, transformation)

# 将数据聚类成3个簇
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# 画出簇分配和簇中心
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm3)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=[0, 1, 2], s=100, linewidth=2, cmap=mglearn.cm3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

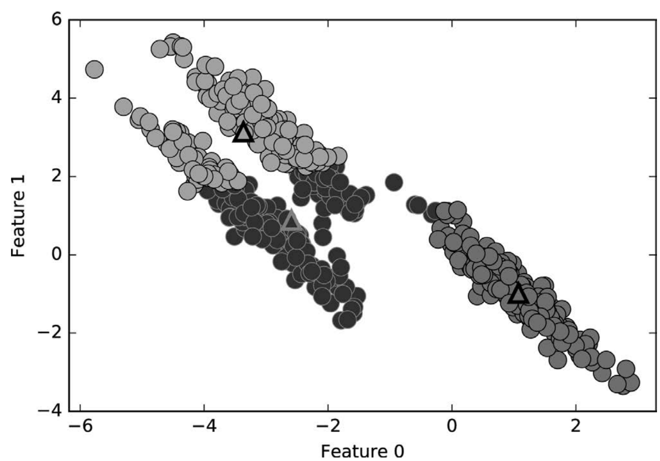


图 3-28：k 均值无法识别非球形簇

如果簇的形状更加复杂，比如我们在第 2 章遇到的 two_moons 数据，那么 k 均值的表现也很差（见图 3-29）：

In[56]:

```
# 生成模拟的two_moons数据（这次的噪声较小）
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# 将数据聚类成2个簇
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# 画出簇分配和簇中心
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm2, s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

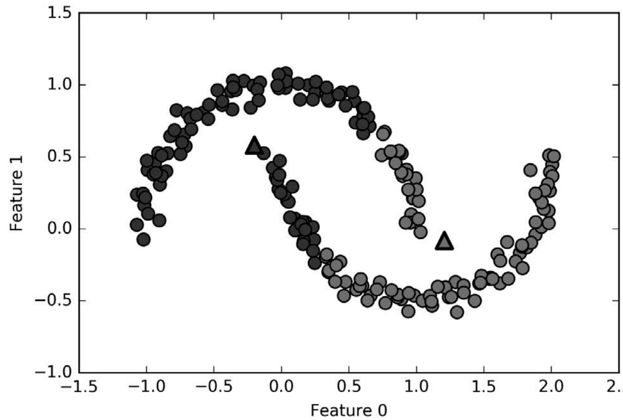


图 3-29：k 均值无法识别具有复杂形状的簇

这里我们希望聚类算法能够发现两个半月形。但利用 k 均值算法是不可能做到这一点的。

2. 矢量量化，或者将k均值看作分解

虽然 k 均值是一种聚类算法，但在 k 均值和分解方法（比如之前讨论过的 PCA 和 NMF）之间存在一些有趣的相似之处。你可能还记得，PCA 试图找到数据中方差最大的方向，而 NMF 试图找到累加的分量，这通常对应于数据的"极值"或"部分"（见图 3-13）。两种方法都试图将数据点表示为一些分量之和。与之相反，k 均值则尝试利用簇中心来表示每个数据点。你可以将其看作仅用一个分量来表示每个数据点，该分量由簇中心给出。这种观点将 k 均值看作是一种分解方法，其中每个点用单一分量来表示，这种观点被称为**矢量量化**（vector quantization）。

我们来并排比较 PCA、NMF 和 k 均值，分别显示提取的分量（图 3-30），以及利用 100 个分量对测试集中人脸的重建（图 3-31）。对于 k 均值，重建就是在训练集中找到的最近的簇中心：

In[57]:

```
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
nmf = NMF(n_components=100, random_state=0)
nmf.fit(X_train)
pca = PCA(n_components=100, random_state=0)
pca.fit(X_train)
kmeans = KMeans(n_clusters=100, random_state=0)
kmeans.fit(X_train)

X_reconstructed_pca = pca.inverse_transform(pca.transform(X_test))
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)
```

In[58]:

```
fig, axes = plt.subplots(3, 5, figsize=(8, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
```

```
fig.suptitle("Extracted Components")
for ax, comp_kmeans, comp_pca, comp_nmf in zip(
        axes.T, kmeans.cluster_centers_, pca.components_, nmf.components_):
    ax[0].imshow(comp_kmeans.reshape(image_shape))
    ax[1].imshow(comp_pca.reshape(image_shape), cmap='viridis')
    ax[2].imshow(comp_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("kmeans")
axes[1, 0].set_ylabel("pca")
axes[2, 0].set_ylabel("nmf")

fig, axes = plt.subplots(4, 5, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(8, 8))
fig.suptitle("Reconstructions")
for ax, orig, rec_kmeans, rec_pca, rec_nmf in zip(
        axes.T, X_test, X_reconstructed_kmeans, X_reconstructed_pca,
        X_reconstructed_nmf):

    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(rec_kmeans.reshape(image_shape))
    ax[2].imshow(rec_pca.reshape(image_shape))
    ax[3].imshow(rec_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("original")
axes[1, 0].set_ylabel("kmeans")
axes[2, 0].set_ylabel("pca")
axes[3, 0].set_ylabel("nmf")
```

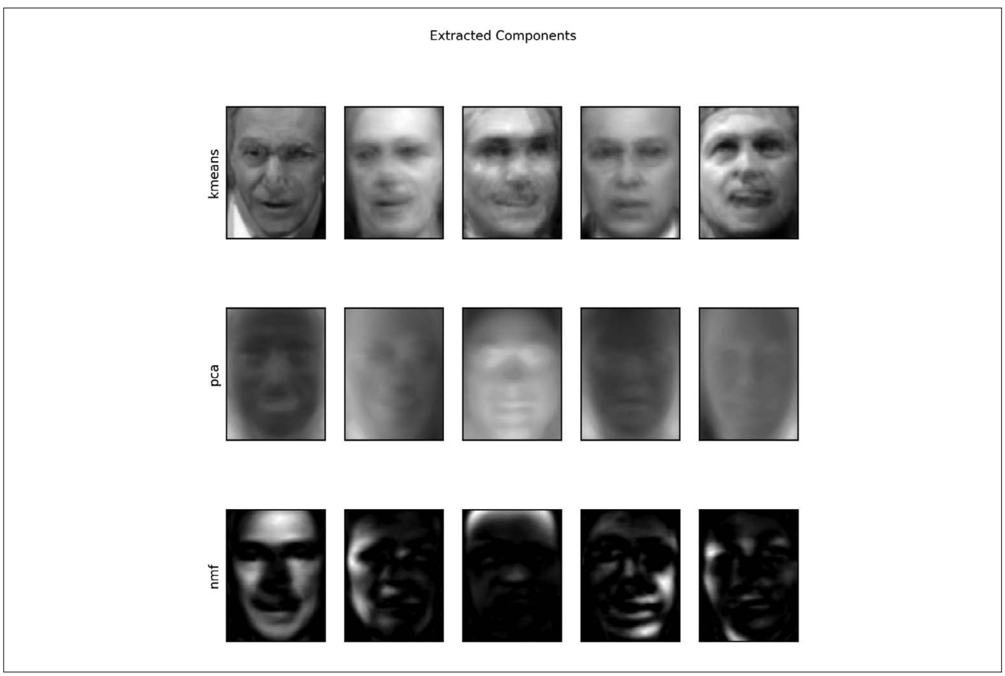


图 3-30：对比 k 均值的簇中心与 PCA 和 NMF 找到的分量

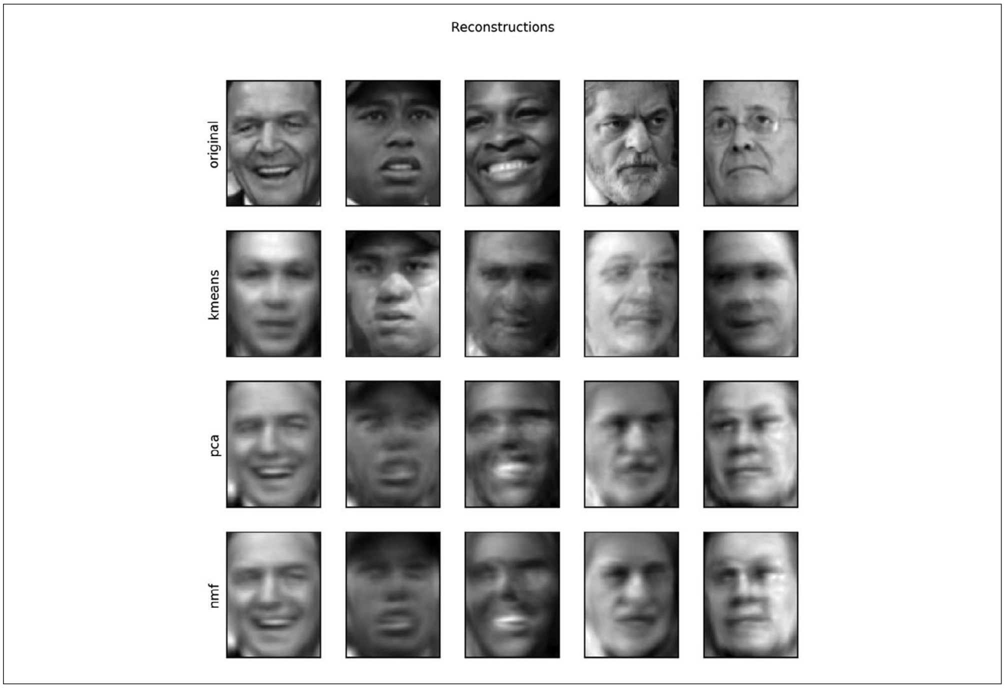


图 3-31：利用 100 个分量（或簇中心）的 k 均值、PCA 和 NMF 的图像重建的对比——k 均值的每张图像中仅使用了一个簇中心

利用 k 均值做矢量量化的一个有趣之处在于，可以用比输入维度更多的簇来对数据进行编码。让我们回到 two_moons 数据。利用 PCA 或 NMF，我们对这个数据无能为力，因为它只有两个维度。使用 PCA 或 NMF 将其降到一维，将会完全破坏数据的结构。但通过使用更多的簇中心，我们可以用 k 均值找到一种更具表现力的表示（见图 3-32）：

In[59]:

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

kmeans = KMeans(n_clusters=10, random_state=0)
kmeans.fit(X)
y_pred = kmeans.predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60, cmap='Paired')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=60,
            marker='^', c=range(kmeans.n_clusters), linewidth=2, cmap='Paired')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
print("Cluster memberships:\n{}".format(y_pred))
```

Out[59]:

```
Cluster memberships:
[9 2 5 4 2 7 9 6 9 6 1 0 2 6 1 9 3 0 3 1 7 6 8 6 8 5 2 7 5 8 9 8 6 5 3 7 0
 9 4 5 0 1 3 5 2 8 9 1 5 6 1 0 7 4 6 3 3 6 3 8 0 4 2 9 6 4 8 2 8 4 0 4 0 5
 6 4 5 9 3 0 7 8 0 7 5 8 9 8 0 7 3 9 7 1 7 2 2 0 4 5 6 7 8 9 4 5 4 1 2 3 1
```

```
8 8 4 9 2 3 7 0 9 9 1 5 8 5 1 9 5 6 7 9 1 4 0 6 2 6 4 7 9 5 5 3 8 1 9 5 6
3 5 0 2 9 3 0 8 6 0 3 3 5 6 3 2 0 2 3 0 2 6 3 4 4 1 5 6 7 1 1 3 2 4 7 2 7
3 8 6 4 1 4 3 9 9 5 1 7 5 8 2]
```

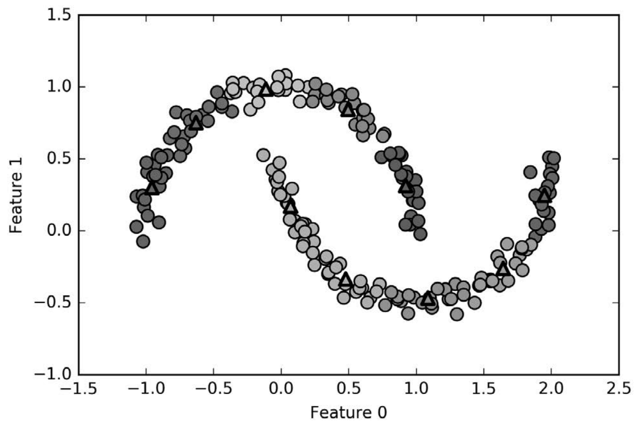


图 3-32：利用 k 均值的许多簇来表示复杂数据集中的变化

我们使用了 10 个簇中心，也就是说，现在每个点都被分配了 0 到 9 之间的一个数字。我们可以将其看作 10 个分量表示的数据（我们有 10 个新特征），只有表示该点对应的簇中心的那个特征不为 0，其他特征均为 0。利用这个 10 维表示，现在可以用线性模型来划分两个半月形，而利用原始的两个特征是不可能做到这一点的。将到每个簇中心的距离作为特征，还可以得到一种表现力更强的数据表示。可以利用 kmeans 的 transform 方法来完成这一点：

In[60]:

```
distance_features = kmeans.transform(X)
print("Distance feature shape: {}".format(distance_features.shape))
print("Distance features:\n{}".format(distance_features))
```

Out[60]:

```
Distance feature shape: (200, 10)
Distance features:
[[ 0.922  1.466  1.14  ...,  1.166  1.039  0.233]
 [ 1.142  2.517  0.12  ...,  0.707  2.204  0.983]
 [ 0.788  0.774  1.749 ...,  1.971  0.716  0.944]
 ...,
 [ 0.446  1.106  1.49  ...,  1.791  1.032  0.812]
 [ 1.39   0.798  1.981 ...,  1.978  0.239  1.058]
 [ 1.149  2.454  0.045 ...,  0.572  2.113  0.882]]
```

k 均值是非常流行的聚类算法，因为它不仅相对容易理解和实现，而且运行速度也相对较快。k 均值可以轻松扩展到大型数据集，scikit-learn 甚至在 MiniBatchKMeans 类中包含了一种更具可扩展性的变体，可以处理非常大的数据集。

k 均值的缺点之一在于，它依赖于随机初始化，也就是说，算法的输出依赖于随机种子。默认情况下，scikit-learn 用 10 种不同的随机初始化将算法运行 10 次，并返回最佳结

果。[4] k 均值还有一个缺点，就是对簇形状的假设的约束性较强，而且还要求指定所要寻找的簇的个数（在现实世界的应用中可能并不知道这个数字）。

接下来，我们将学习另外两种聚类算法，它们都在某些方面对这些性质做了改进。

3.5.2　凝聚聚类

凝聚聚类（agglomerative clustering）指的是许多基于相同原则构建的聚类算法，这一原则是：算法首先声明每个点是自己的簇，然后合并两个最相似的簇，直到满足某种停止准则为止。`scikit-learn` 中实现的停止准则是簇的个数，因此相似的簇被合并，直到仅剩下指定个数的簇。还有一些**链接**（linkage）准则，规定如何度量“最相似的簇”。这种度量总是定义在两个现有的簇之间。

`scikit-learn` 中实现了以下三种选项。

`ward`

　默认选项。`ward` 挑选两个簇来合并，使得所有簇中的方差增加最小。这通常会得到大小差不多相等的簇。

`average`

　`average` 链接将簇中所有点之间平均距离最小的两个簇合并。

`complete`

　`complete` 链接（也称为最大链接）将簇中点之间最大距离最小的两个簇合并。

`ward` 适用于大多数数据集，在我们的例子中将使用它。如果簇中的成员个数非常不同（比如其中一个比其他所有都大得多），那么 `average` 或 `complete` 可能效果更好。

图 3-33 给出了在一个二维数据集上的凝聚聚类过程，要寻找三个簇。

In[61]:

```
mglearn.plots.plot_agglomerative_algorithm()
```

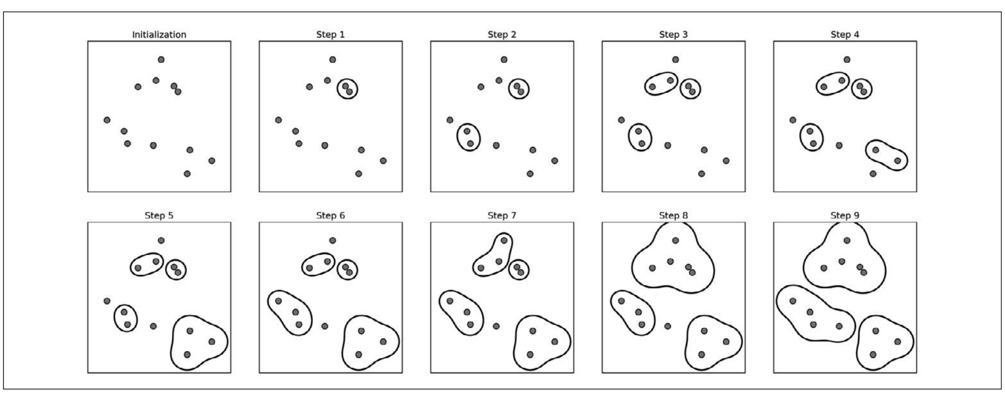


图 3-33：凝聚聚类用迭代的方式合并两个最近的簇

注 4：在这种情况下，“最佳”的意思是簇的方差之和最小。

最开始，每个点自成一簇。然后在每一个步骤中，相距最近的两个簇被合并。在前四个步骤中，选出两个单点簇并将其合并成两点簇。在步骤 5（Step 5）中，其中一个两点簇被扩展到三个点，以此类推。在步骤 9（Step 9）中，只剩下 3 个簇。由于我们指定寻找 3 个簇，因此算法结束。

我们来看一下凝聚聚类对我们这里使用的简单三簇数据的效果如何。由于算法的工作原理，凝聚算法不能对新数据点做出预测。因此 `AgglomerativeClustering` 没有 `predict` 方法。为了构造模型并得到训练集上簇的成员关系，可以改用 `fit_predict` 方法。[5] 结果如图 3-34 所示。

In[62]:

```
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)

agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

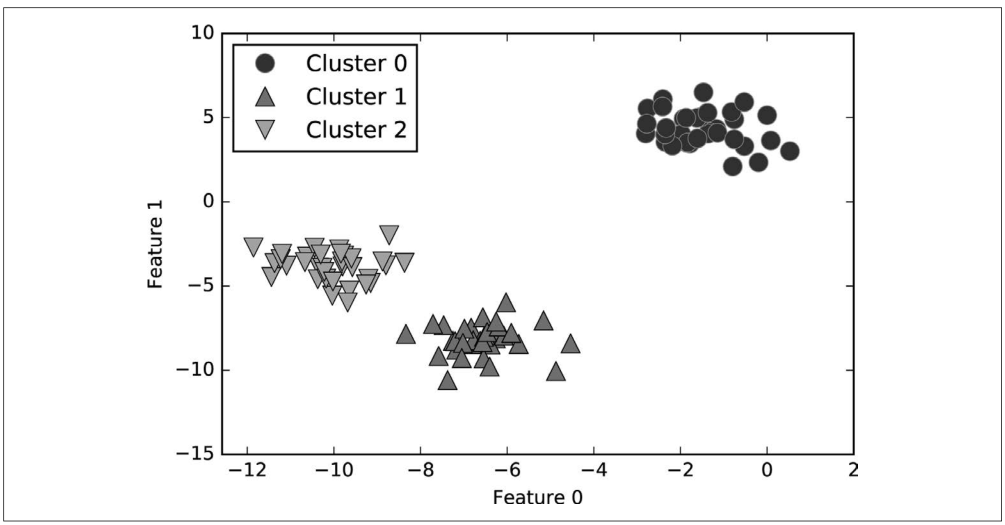


图 3-34：使用 3 个簇的凝聚聚类的簇分配

正如所料，算法完美地完成了聚类。虽然凝聚聚类的 `scikit-learn` 实现需要你指定希望算法找到的簇的个数，但凝聚聚类方法为选择正确的个数提供了一些帮助，我们将在下面讨论。

1. 层次聚类与树状图

凝聚聚类生成了所谓的**层次聚类**（hierarchical clustering）。聚类过程迭代进行，每个点都

注 5：我们也可以使用 `labels_` 属性，正如 k 均值所做的那样。

从一个单点簇变为属于最终的某个簇。每个中间步骤都提供了数据的一种聚类（簇的个数也不相同）。有时候，同时查看所有可能的聚类是有帮助的。下一个例子（图 3-35）叠加显示了图 3-33 中所有可能的聚类，有助于深入了解每个簇如何分解为较小的簇：

In[63]:

```
mglearn.plots.plot_agglomerative()
```

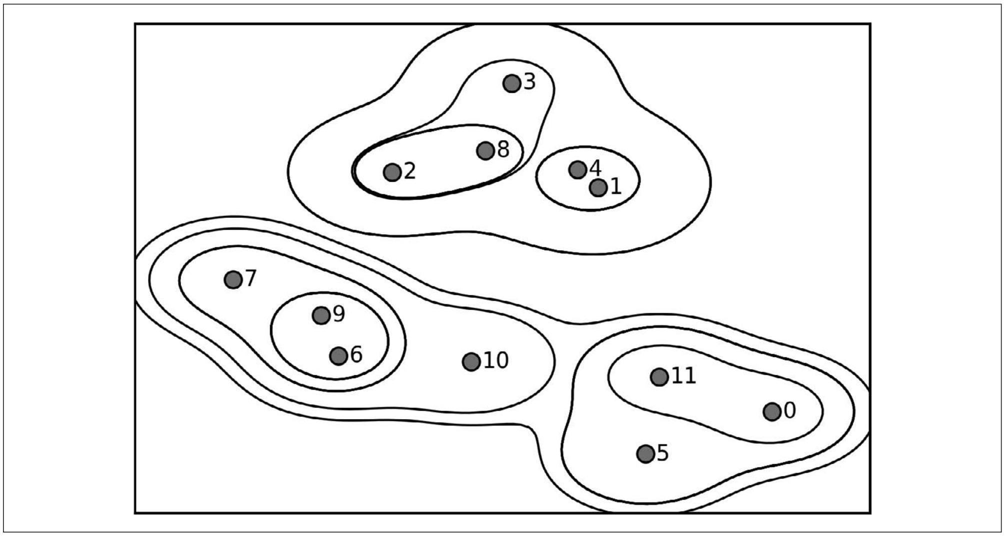


图 3-35：凝聚聚类生成的层次化的簇分配（用线表示）以及带有编号的数据点（参见图 3-36）

虽然这种可视化为层次聚类提供了非常详细的视图，但它依赖于数据的二维性质，因此不能用于具有两个以上特征的数据集。但还有另一个将层次聚类可视化的工具，叫作**树状图**（dendrogram），它可以处理多维数据集。

不幸的是，目前 `scikit-learn` 没有绘制树状图的功能。但你可以利用 SciPy 轻松生成树状图。SciPy 的聚类算法接口与 `scikit-learn` 的聚类算法稍有不同。SciPy 提供了一个函数，接受数据数组 `X` 并计算出一个**链接数组**（linkage array），它对层次聚类的相似度进行编码。然后我们可以将这个链接数组提供给 scipy 的 `dendrogram` 函数来绘制树状图（图 3-36）。

In[64]:

```
# 从SciPy中导入dendrogram函数和ward聚类函数
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
# 将ward聚类应用于数据数组X
# SciPy的ward函数返回一个数组，指定执行凝聚聚类时跨越的距离
linkage_array = ward(X)
# 现在为包含簇之间距离的linkage_array绘制树状图
dendrogram(linkage_array)

# 在树中标记划分成两个簇或三个簇的位置
ax = plt.gca()
```

```
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')

ax.text(bounds[1], 7.25, ' two clusters', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, ' three clusters', va='center', fontdict={'size': 15})
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```

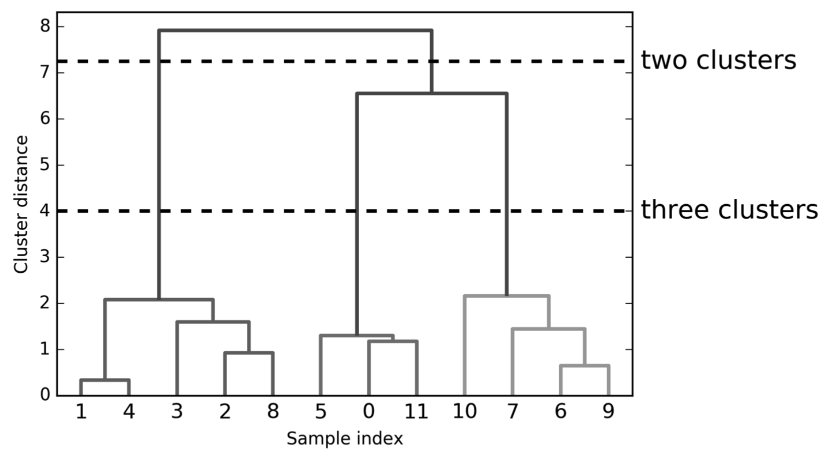


图 3-36：图 3-35 中聚类的树状图（用线表示划分成两个簇和三个簇）

树状图在底部显示数据点（编号从 0 到 11）。然后以这些点（表示单点簇）作为叶节点绘制一棵树，每合并两个簇就添加一个新的父节点。

从下往上看，数据点 1 和 4 首先被合并（正如你在图 3-33 中所见）。接下来，点 6 和 9 被合并为一个簇，以此类推。在顶层有两个分支，一个由点 11、0、5、10、7、6 和 9 组成，另一个由点 1、4、3、2 和 8 组成。这对应于图中左侧两个最大的簇。

树状图的 y 轴不仅说明凝聚算法中两个簇何时合并，每个分支的长度还表示被合并的簇之间的距离。在这张树状图中，最长的分支是用标记为“three clusters”（三个簇）的虚线表示的三条线。它们是最长的分支，这表示从三个簇到两个簇的过程中合并了一些距离非常远的点。我们在图像上方再次看到这一点，将剩下的两个簇合并为一个簇也需要跨越相对较大的距离。

不幸的是，凝聚聚类仍然无法分离像 `two_moons` 数据集这样复杂的形状。但我们要学习的下一个算法 DBSCAN 可以解决这个问题。

3.5.3 DBSCAN

另一个非常有用的聚类算法是 DBSCAN（density-based spatial clustering of applications with noise，即“具有噪声的基于密度的空间聚类应用”）。DBSCAN 的主要优点是它不需要用

户**先验**地设置簇的个数，可以划分具有复杂形状的簇，还可以找出不属于任何簇的点。DBSCAN 比凝聚聚类和 k 均值稍慢，但仍可以扩展到相对较大的数据集。

DBSCAN 的原理是识别特征空间的"拥挤"区域中的点，在这些区域中许多数据点靠近在一起。这些区域被称为特征空间中的**密集**（dense）区域。DBSCAN 背后的思想是，簇形成数据的密集区域，并由相对较空的区域分隔开。

在密集区域内的点被称为**核心样本**（core sample，或核心点），它们的定义如下。DBSCAN 有两个参数：`min_samples` 和 `eps`。如果在距一个给定数据点 `eps` 的距离内至少有 `min_samples` 个数据点，那么这个数据点就是核心样本。DBSCAN 将彼此距离小于 `eps` 的核心样本放到同一个簇中。

算法首先任意选取一个点，然后找到到这个点的距离小于等于 `eps` 的所有的点。如果距起始点的距离在 `eps` 之内的数据点个数小于 `min_samples`，那么这个点被标记为**噪声**（noise），也就是说它不属于任何簇。如果距离在 `eps` 之内的数据点个数大于 `min_samples`，则这个点被标记为核心样本，并被分配一个新的簇标签。然后访问该点的所有邻居（在距离 `eps` 以内）。如果它们还没有被分配一个簇，那么就将刚刚创建的新的簇标签分配给它们。如果它们是核心样本，那么就依次访问其邻居，以此类推。簇逐渐增大，直到在簇的 `eps` 距离内没有更多的核心样本为止。然后选取另一个尚未被访问过的点，并重复相同的过程。

最后，一共有三种类型的点：核心点、与核心点的距离在 `eps` 之内的点（叫作**边界点**，boundary point）和噪声。如果 DBSCAN 算法在特定数据集上多次运行，那么核心点的聚类始终相同，同样的点也始终被标记为噪声。但边界点可能与不止一个簇的核心样本相邻。因此，边界点所属的簇依赖于数据点的访问顺序。一般来说只有很少的边界点，这种对访问顺序的轻度依赖并不重要。

我们将 DBSCAN 应用于演示凝聚聚类的模拟数据集。与凝聚聚类类似，DBSCAN 也不允许对新的测试数据进行预测，所以我们将使用 `fit_predict` 方法来执行聚类并返回簇标签。

In[65]:

```
from sklearn.cluster import DBSCAN
X, y = make_blobs(random_state=0, n_samples=12)

dbscan = DBSCAN()
clusters = dbscan.fit_predict(X)
print("Cluster memberships:\n{}".format(clusters))
```

Out[65]:

```
Cluster memberships:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

如你所见，所有数据点都被分配了标签 `-1`，这代表噪声。这是 `eps` 和 `min_samples` 默认参数设置的结果，对于小型的玩具数据集并没有调节这些参数。`min_samples` 和 `eps` 取不同值时的簇分类如下所示，其可视化结果见图 3-37。

In[66]:

```
mglearn.plots.plot_dbscan()
```

Out[66]:

```
min_samples: 2 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]
min_samples: 2 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 2 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 2 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 3 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]
min_samples: 3 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 3 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 3 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 5 eps: 1.000000  cluster: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
min_samples: 5 eps: 1.500000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1 -1]
min_samples: 5 eps: 2.000000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1 -1]
min_samples: 5 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
```

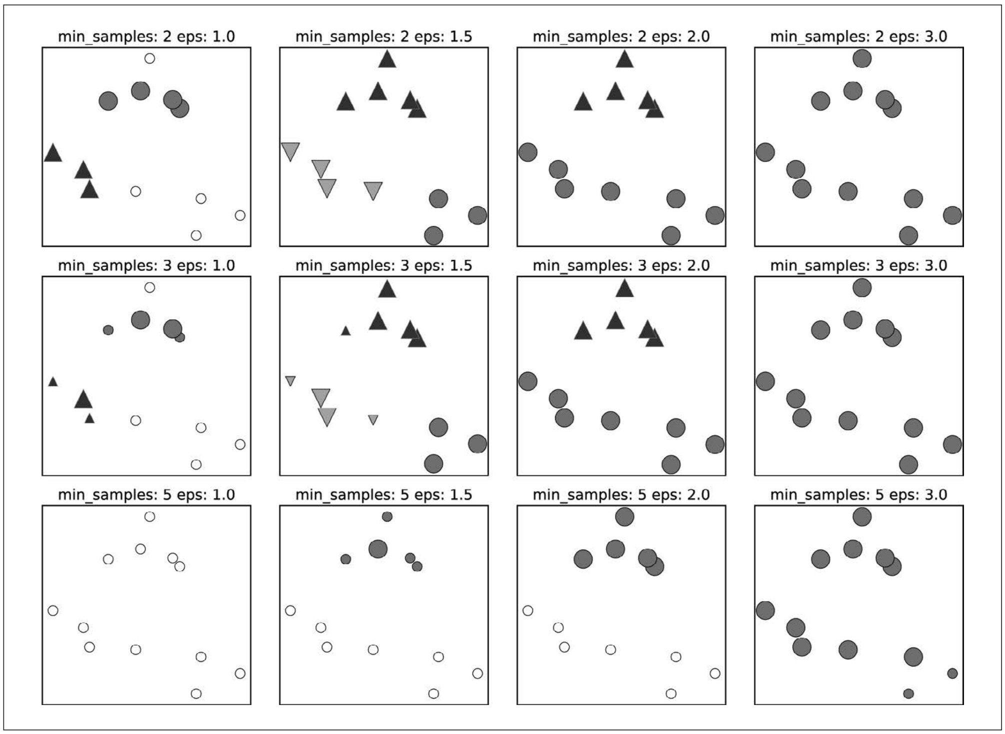


图 3-37：在 min_samples 和 eps 参数不同取值的情况下，DBSCAN 找到的簇分配

在这张图中，属于簇的点是实心的，而噪声点则显示为空心的。核心样本显示为较大的标记，而边界点则显示为较小的标记。增大 eps（在图中从左到右），更多的点会被包含在一个簇中。这让簇变大，但可能也会导致多个簇合并成一个。增大 min_samples（在图中从上到下），核心点会变得更少，更多的点被标记为噪声。

参数 eps 在某种程度上更加重要，因为它决定了点与点之间“接近”的含义。将 eps 设置得非常小，意味着没有点是核心样本，可能会导致所有点都被标记为噪声。将 eps 设置得非常大，可能会导致所有点形成单个簇。

设置 min_samples 主要是为了判断稀疏区域内的点被标记为异常值还是形成自己的簇。如果增大 min_samples，任何一个包含少于 min_samples 个样本的簇现在将被标记为噪声。因此，min_samples 决定簇的最小尺寸。在图 3-37 中 eps=1.5 时，从 min_samples=3 到 min_samples=5，你可以清楚地看到这一点。min_samples=3 时有三个簇：一个包含 4 个点，一个包含 5 个点，一个包含 3 个点。min_samples=5 时，两个较小的簇（分别包含 3 个点和 4 个点）现在被标记为噪声，只保留包含 5 个样本的簇。

虽然 DBSCAN 不需要显式地设置簇的个数，但设置 eps 可以隐式地控制找到的簇的个数。使用 StandardScaler 或 MinMaxScaler 对数据进行缩放之后，有时会更容易找到 eps 的较好取值，因为使用这些缩放技术将确保所有特征具有相似的范围。

图 3-38 展示了在 two_moons 数据集上运行 DBSCAN 的结果。利用默认设置，算法找到了两个半圆形并将其分开：

In[67]:

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# 将数据缩放成平均值为0、方差为1
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

dbscan = DBSCAN()
clusters = dbscan.fit_predict(X_scaled)
# 绘制簇分配
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm2, s=60)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

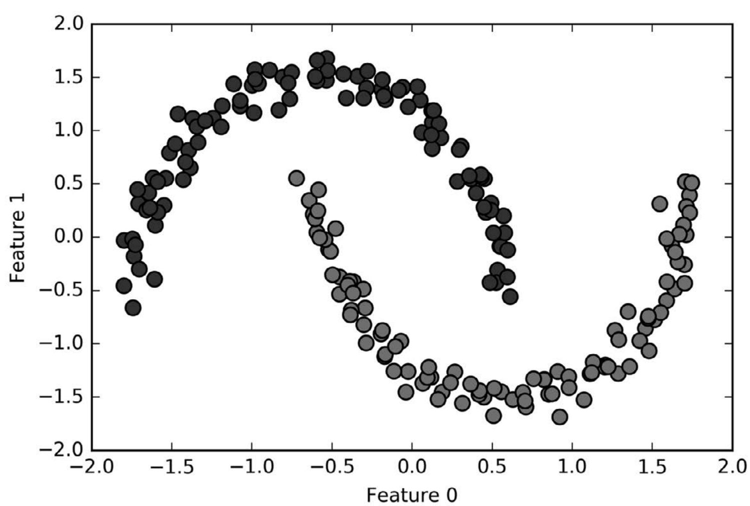


图 3-38：利用默认值 eps=0.5 的 DBSCAN 找到的簇分配

由于算法找到了我们想要的簇的个数（2 个），因此参数设置的效果似乎很好。如果将 eps 减小到 0.2（默认值为 0.5），我们将会得到 8 个簇，这显然太多了。将 eps 增大到 0.7 则会导致只有一个簇。

在使用 DBSCAN 时，你需要谨慎处理返回的簇分配。如果使用簇标签对另一个数据进行索引，那么使用 -1 表示噪声可能会产生意料之外的结果。

3.5.4 聚类算法的对比与评估

在应用聚类算法时，其挑战之一就是很难评估一个算法的效果好坏，也很难比较不同算法的结果。在讨论完 k 均值、凝聚聚类和 DBSCAN 背后的算法之后，下面我们将在一些现实世界的数据集上比较它们。

1. 用真实值评估聚类

有一些指标可用于评估聚类算法相对于真实聚类的结果，其中最重要的是**调整 rand 指数**（adjusted rand index，ARI）和**归一化互信息**（normalized mutual information，NMI），二者都给出了定量的度量，其最佳值为 1，0 表示不相关的聚类（虽然 ARI 可以取负值）。

下面我们使用 ARI 来比较 k 均值、凝聚聚类和 DBSCAN 算法。为了对比，我们还添加了将点随机分配到两个簇中的图像（见图 3-39）。

In[68]:

```
from sklearn.metrics.cluster import adjusted_rand_score
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# 将数据缩放成平均值为0、方差为1
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                         subplot_kw={'xticks': (), 'yticks': ()})

# 列出要使用的算法
algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

# 创建一个随机的簇分配，作为参考
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# 绘制随机分配
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment - ARI: {:.2f}".format(
        adjusted_rand_score(y, random_clusters)))

for ax, algorithm in zip(axes[1:], algorithms):
    # 绘制簇分配和簇中心
    clusters = algorithm.fit_predict(X_scaled)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters,
```

```
               cmap=mglearn.cm3, s=60)
    ax.set_title("{} - ARI: {:.2f}".format(algorithm.__class__.__name__,
                                           adjusted_rand_score(y, clusters)))
```

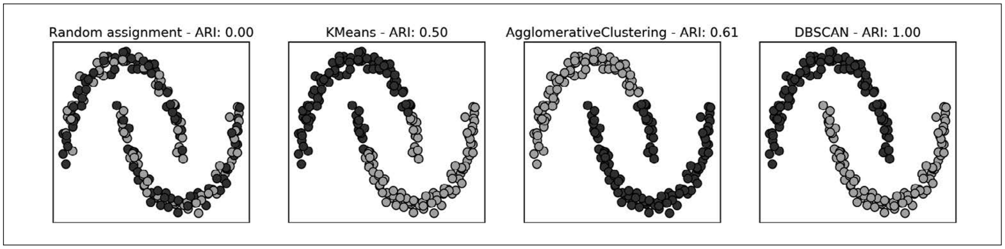


图 3-39：利用监督 ARI 分数在 two_moons 数据集上比较随机分配、k 均值、凝聚聚类和 DBSCAN

调整 rand 指数给出了符合直觉的结果，随机簇分配的分数为 0，而 DBSCAN（完美地找到了期望中的聚类）的分数为 1。

用这种方式评估聚类时，一个常见的错误是使用 accuracy_score 而不是 adjusted_rand_score、normalized_mutual_info_score 或其他聚类指标。使用精度的问题在于，它要求分配的簇标签与真实值完全匹配。但簇标签本身毫无意义——唯一重要的是哪些点位于同一个簇中。

In[69]:

```
from sklearn.metrics import accuracy_score

# 这两种点标签对应于相同的聚类
clusters1 = [0, 0, 1, 1, 0]
clusters2 = [1, 1, 0, 0, 1]
# 精度为0，因为二者标签完全不同
print("Accuracy: {:.2f}".format(accuracy_score(clusters1, clusters2)))
# 调整rand分数为1，因为二者聚类完全相同
print("ARI: {:.2f}".format(adjusted_rand_score(clusters1, clusters2)))
```

Out[69]:

```
Accuracy: 0.00
ARI: 1.00
```

2. 在没有真实值的情况下评估聚类

我们刚刚展示了一种评估聚类算法的方法，但在实践中，使用诸如 ARI 之类的指标有一个很大的问题。在应用聚类算法时，通常没有真实值来比较结果。如果我们知道了数据的正确聚类，那么可以使用这一信息构建一个监督模型（比如分类器）。因此，使用类似 ARI 和 NMI 的指标通常仅有助于开发算法，但对评估应用是否成功没有帮助。

有一些聚类的评分指标不需要真实值，比如**轮廓系数**（silhouette coeffcient）。但它们在实践中的效果并不好。轮廓分数计算一个簇的紧致度，其值越大越好，最高分数为 1。虽然紧致的簇很好，但紧致度不允许复杂的形状。

下面是一个例子，利用轮廓分数在 two_moons 数据集上比较 k 均值、凝聚聚类和 DBSCAN（图 3-40）：

In[70]:

```
from sklearn.metrics.cluster import silhouette_score

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# 将数据缩放成平均值为0、方差为1
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                         subplot_kw={'xticks': (), 'yticks': ()})

# 创建一个随机的簇分配，作为参考
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# 绘制随机分配
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
    cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment: {:.2f}".format(
    silhouette_score(X_scaled, random_clusters)))

algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

for ax, algorithm in zip(axes[1:], algorithms):
    clusters = algorithm.fit_predict(X_scaled)
    # 绘制簇分配和簇中心
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm3,
               s=60)
    ax.set_title("{} : {:.2f}".format(algorithm.__class__.__name__,
                                      silhouette_score(X_scaled, clusters)))
```

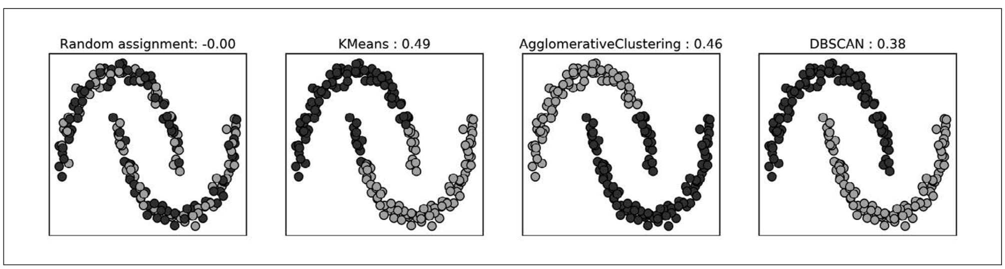


图 3-40：利用无监督的轮廓分数在 two_moons 数据集上比较随机分配、k 均值、凝聚聚类和 DBSCAN（更符合直觉的 DBSCAN 的轮廓分数低于 k 均值找到的分配）

如你所见，k 均值的轮廓分数最高，尽管我们可能更喜欢 DBSCAN 的结果。对于评估聚类，稍好的策略是使用**基于鲁棒性的**（robustness-based）聚类指标。这种指标先向数据中添加一些噪声，或者使用不同的参数设定，然后运行算法，并对结果进行比较。其思想是，如果许多算法参数和许多数据扰动返回相同的结果，那么它很可能是可信的。不幸的是，在写作本书时，scikit-learn 还没有实现这一策略。

即使我们得到一个鲁棒性很好的聚类或者非常高的轮廓分数，但仍然不知道聚类中是否有

任何语义含义，或者聚类是否反映了数据中我们感兴趣的某个方面。我们回到人脸图像的例子。我们希望找到类似人脸的分组，比如男人和女人、老人和年轻人，或者有胡子的人和没胡子的人。假设我们将数据分为两个簇，关于哪些点应该被聚类在一起，所有算法的结果一致。我们仍不知道找到的簇是否以某种方式对应于我们感兴趣的概念。算法找到的可能是侧视图和正面视图、夜间拍摄的照片和白天拍摄的照片，或者 iPhone 拍摄的照片和安卓手机拍摄的照片。要想知道聚类是否对应于我们感兴趣的内容，唯一的办法就是对簇进行人工分析。

3. 在人脸数据集上比较算法

我们将 k 均值、DBSCAN 和凝聚聚类算法应用于 Wild 数据集中的 Labeled Faces，并查看它们是否找到了有趣的结构。我们将使用数据的特征脸表示，它由包含 100 个成分的 `PCA(whiten=True)` 生成：

In[71]:

```
# 从lfw数据中提取特征脸，并对数据进行变换
from sklearn.decomposition import PCA
pca = PCA(n_components=100, whiten=True, random_state=0)
pca.fit_transform(X_people)
X_pca = pca.transform(X_people)
```

我们之前见到，与原始像素相比，这是对人脸图像的一种语义更强的表示。它的计算速度也更快。这里有一个很好的练习，就是在原始数据上运行下列实验，不要用 PCA，并观察你是否能找到类似的簇。

用 DBSCAN 分析人脸数据集。我们首先应用刚刚讨论过的 DBSCAN：

In[72]:

```
# 应用默认参数的DBSCAN
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[72]:

```
Unique labels: [-1]
```

我们看到，所有返回的标签都是 –1，因此所有数据都被 DBSCAN 标记为“噪声”。我们可以改变两个参数来改进这一点：第一，我们可以增大 `eps`，从而扩展每个点的邻域；第二，我们可以减小 `min_samples`，从而将更小的点组视为簇。我们首先尝试改变 `min_samples`：

In[73]:

```
dbscan = DBSCAN(min_samples=3)
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[73]:

```
Unique labels: [-1]
```

即使仅考虑由三个点构成的组，所有点也都被标记为噪声。因此我们需要增大 `eps`：

In[74]:

```
dbscan = DBSCAN(min_samples=3, eps=15)
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[74]:

```
Unique labels: [-1  0]
```

使用更大的 eps（其值为 15），我们只得到了单一簇和噪声点。我们可以利用这一结果找出"噪声"相对于其他数据的形状。为了进一步理解发生的事情，我们查看有多少点是噪声，有多少点在簇内：

In[75]:

```
# 计算所有簇中的点数和噪声中的点数。
# bincount不允许负值，所以我们需要加1。
# 结果中的第一个数字对应于噪声点。
print("Number of points per cluster: {}".format(np.bincount(labels + 1)))
```

Out[75]:

```
Number of points per cluster: [  27 2036]
```

噪声点非常少——只有 27 个，因此我们可以查看所有的噪声点（见图 3-41）：

In[76]:

```
noise = X_people[labels==-1]

fig, axes = plt.subplots(3, 9, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(12, 4))
for image, ax in zip(noise, axes.ravel()):
    ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
```



图 3-41：人脸数据集中被 DBSCAN 标记为噪声的样本

将这些图像与图 3-7 中随机选择的人脸图像样本进行比较，我们可以猜测它们被标记为噪声的原因：第 1 行第 5 张图像显示一个人正在用玻璃杯喝水，还有人戴帽子的图像，在最后一张图像中，人脸前面有一只手。其他图像都包含奇怪的角度，或者太近或太宽的剪切。

这种类型的分析——尝试找出“奇怪的那一个”——被称为**异常值检测**（outlier detection）。如果这是一个真实的应用，那么我们可能会尝试更好地裁切图像，以得到更加均匀的数据。对于照片中的人有时戴着帽子、喝水或在面前举着某物，我们能做的事情很少。但需要知道它们是数据中存在的问题，我们应用任何算法都需要解决这些问题。

如果我们想要找到更有趣的簇，而不是一个非常大的簇，那么需要将 `eps` 设置得更小，取值在 `15` 和 `0.5`（默认值）之间。我们来看一下 `eps` 不同取值对应的结果：

In[77]:

```
for eps in [1, 3, 5, 7, 9, 11, 13]:
    print("\neps={}".format(eps))
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Clusters present: {}".format(np.unique(labels)))
    print("Cluster sizes: {}".format(np.bincount(labels + 1)))
```

Out[77]:

```
eps=1
Clusters present: [-1]
Cluster sizes: [2063]

eps=3
Clusters present: [-1]
Cluster sizes: [2063]

eps=5
Clusters present: [-1]
Cluster sizes: [2063]

eps=7
Clusters present: [-1  0  1  2  3  4  5  6  7  8  9 10 11 12]
Cluster sizes: [2006  4  6  6  6  9  3  3  4  3  3  3  3  4]

eps=9
Clusters present: [-1  0  1  2]
Cluster sizes: [1269  788    3    3]

eps=11
Clusters present: [-1  0]
Cluster sizes: [ 430 1633]

eps=13
Clusters present: [-1  0]
Cluster sizes: [ 112 1951]
```

对于较小的 `eps`，所有点都被标记为噪声。`eps=7` 时，我们得到许多噪声点和许多较小的簇。`eps=9` 时，我们仍得到许多噪声点，但我们得到了一个较大的簇和一些较小的簇。从 `eps=11` 开始，我们仅得到一个较大的簇和噪声。

有趣的是，较大的簇从来没有超过一个。最多有一个较大的簇包含大多数点，还有一些较小的簇。这表示数据中没有两类或三类非常不同的人脸图像，而是所有图像或多或少地都与其他图像具有相同的相似度（或不相似度）。

eps=7 的结果看起来最有趣，它有许多较小的簇。我们可以通过将 13 个较小的簇中的点全部可视化来深入研究这一聚类（图 3-42）：

In[78]:

```
dbscan = DBSCAN(min_samples=3, eps=7)
labels = dbscan.fit_predict(X_pca)

for cluster in range(max(labels) + 1):
    mask = labels == cluster
    n_images =  np.sum(mask)
    fig, axes = plt.subplots(1, n_images, figsize=(n_images * 1.5, 4),
                             subplot_kw={'xticks': (), 'yticks': ()})
    for image, label, ax in zip(X_people[mask], y_people[mask], axes):

        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1])
```

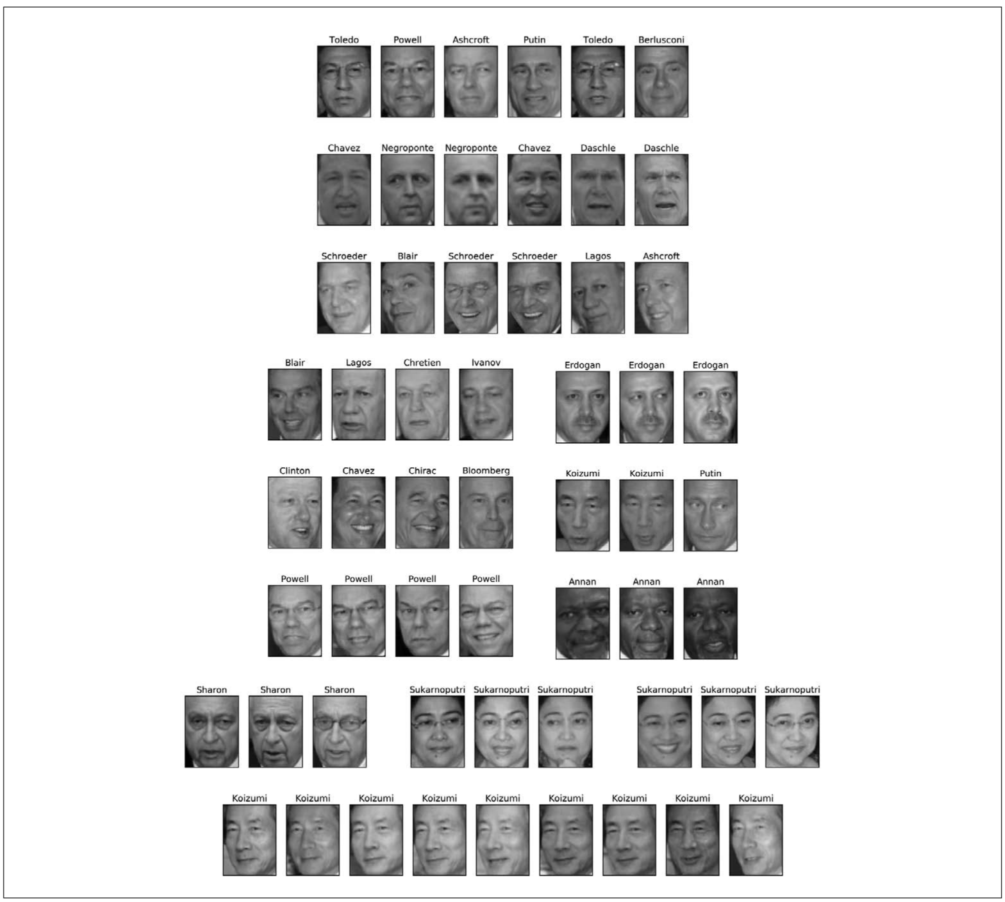


图 3-42：eps=7 的 DBSCAN 找到的簇

有一些簇对应于（这个数据集中）脸部非常不同的人，比如 Sharon（沙龙）或 Koizumi

（小泉）。在每个簇内，人脸方向和面部表情也是固定的。有些簇中包含多个人的面孔，但他们的方向和表情都相似。

这就是我们将 DBSCAN 算法应用于人脸数据集的分析结论。如你所见，我们这里进行了人工分析，不同于监督学习中基于 R^2 分数或精度的更为自动化的搜索方法。

下面我们将继续应用 k 均值和凝聚聚类。

用 k 均值分析人脸数据集。我们看到，利用 DBSCAN 无法创建多于一个较大的簇。凝聚聚类和 k 均值更可能创建均匀大小的簇，但我们需要设置簇的目标个数。我们可以将簇的数量设置为数据集中的已知人数，虽然无监督聚类算法不太可能完全找到它们。相反，我们可以首先设置一个比较小的簇的数量，比如 10 个，这样我们可以分析每个簇：

In[79]:

```
# 用k均值提取簇
km = KMeans(n_clusters=10, random_state=0)
labels_km = km.fit_predict(X_pca)
print("Cluster sizes k-means: {}".format(np.bincount(labels_km)))
```

Out[79]:

```
Cluster sizes k-means: [269 128 170 186 386 222 237  64 253 148]
```

如你所见，k 均值聚类将数据划分为大小相似的簇，其大小在 64 和 386 之间。这与 DBSCAN 的结果非常不同。

我们可以通过将簇中心可视化来进一步分析 k 均值的结果（图 3-43）。由于我们是在 PCA 生成的表示中进行聚类，因此我们需要使用 pca.inverse_transform 将簇中心旋转回到原始空间并可视化：

In[80]:

```
fig, axes = plt.subplots(2, 5, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(12, 4))
for center, ax in zip(km.cluster_centers_, axes.ravel()):
    ax.imshow(pca.inverse_transform(center).reshape(image_shape),
              vmin=0, vmax=1)
```

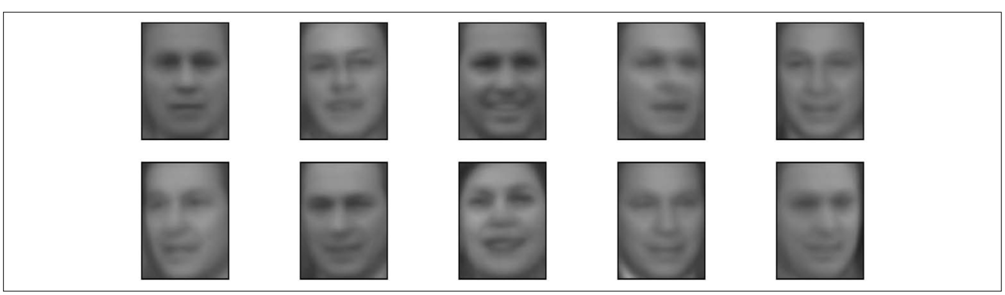


图 3-43：将簇的数量设置为 10 时，k 均值找到的簇中心

k 均值找到的簇中心是非常平滑的人脸。这并不奇怪，因为每个簇中心都是 64 到 386 张人脸图像的平均。使用降维的 PCA 表示，可以增加图像的平滑度（对比图 3-11 中利用 100

个 PCA 维度重建的人脸）。聚类似乎捕捉到人脸的不同方向、不同表情（第 3 个簇中心似乎显示的是一张笑脸），以及是否有衬衫领子（见倒数第二个簇中心）。

图 3-44 给出了更详细的视图，我们对每个簇中心给出了簇中 5 张最典型的图像（该簇中与簇中心距离最近的图像）与 5 张最不典型的图像（该簇中与簇中心距离最远的图像）：

In[81]:

```
mglearn.plots.plot_kmeans_faces(km, pca, X_pca, X_people,
                                y_people, people.target_names)
```



图 3-44：k 均值为每个簇找到的样本图像——簇中心在最左边，然后是五个距中心最近的点，然后是五个距该簇距中心最远的点

图 3-44 证实了我们认为第 3 个簇是笑脸的直觉，也证实了其他簇中方向的重要性。不过“非典型的”点与簇中心不太相似，而且它们的分配似乎有些随意。这可以归因于以下事实：k 均值对所有数据点进行划分，不像 DBSCAN 那样具有“噪声”点的概念。利用更多数量的簇，算法可以找到更细微的区别。但添加更多的簇会使得人工检查更加困难。

用凝聚聚类分析人脸数据集。下面我们来看一下凝聚聚类的结果：

In[82]:

```
# 用ward凝聚聚类提取簇
agglomerative = AgglomerativeClustering(n_clusters=10)
labels_agg = agglomerative.fit_predict(X_pca)
print("Cluster sizes agglomerative clustering: {}".format(
    np.bincount(labels_agg)))
```

Out[82]:

```
Cluster sizes agglomerative clustering: [255 623  86 102 122 199 265  26 230 155]
```

凝聚聚类生成的也是大小相近的簇，其大小在 26 和 623 之间。这比 k 均值生成的簇更不均匀，但比 DBSCAN 生成的簇要更加均匀。

我们可以通过计算 ARI 来度量凝聚聚类和 k 均值给出的两种数据划分是否相似：

In[83]:

```
print("ARI: {:.2f}".format(adjusted_rand_score(labels_agg, labels_km)))
```

Out[83]:

```
ARI: 0.13
```

ARI 只有 0.13，说明 `labels_agg` 和 `labels_km` 这两种聚类的共同点很少。这并不奇怪，原因在于以下事实：对于 k 均值，远离簇中心的点似乎没有什么共同点。

下面，我们可能会想要绘制树状图（图 3-45）。我们将限制图中树的深度，因为如果分支到 2063 个数据点，图像将密密麻麻无法阅读：

In[84]:

```
linkage_array = ward(X_pca)
# 现在我们为包含簇之间距离的linkage_array绘制树状图
plt.figure(figsize=(20, 5))
dendrogram(linkage_array, p=7, truncate_mode='level', no_labels=True)
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```

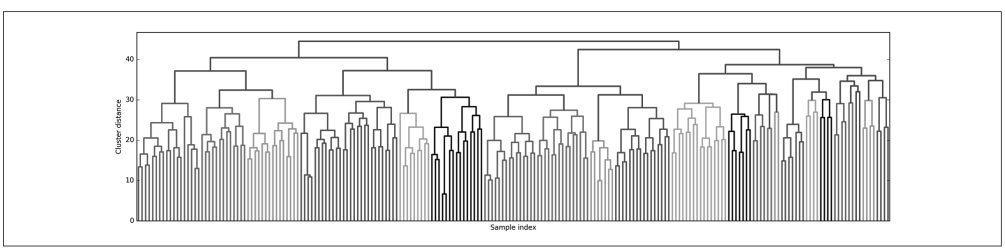


图 3-45：凝聚聚类在人脸数据集上的树状图

要想创建 10 个簇，我们在顶部有 10 条竖线的位置将树横切。在图 3-36 所示的玩具数据的树状图中，你可以从分支的长度中看出，两个或三个簇就可以很好地划分数据。对于人脸数据而言，似乎没有非常自然的切割点。有一些分支代表更为不同的组，但似乎没有一个特别合适的簇的数量。这并不奇怪，因为 DBSCAN 的结果是试图将所有的点都聚

类在一起。

我们将 10 个簇可视化，正如之前对 k 均值所做的那样（图 3-46）。请注意，在凝聚聚类中没有簇中心的概念（虽然我们计算平均值），我们只是给出了每个簇的前几个点。我们在第一张图像的左侧给出了每个簇中的点的数量：

In[85]:

```
n_clusters = 10
for cluster in range(n_clusters):
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 10, subplot_kw={'xticks': (), 'yticks': ()},
                             figsize=(15, 8))
    axes[0].set_ylabel(np.sum(mask))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
```



图 3-46：In[82] 生成的簇中的随机图像——每一行对应一个簇，左侧的数字表示每个簇中图像的数量

虽然某些簇似乎具有语义上的主题，但许多簇都太大而实际上很难是均匀的。为了得到更加均匀的簇，我们可以再次运行算法，这次使用 40 个簇，并挑选出一些特别有趣的簇（图 3-47）：

In[86]:

```
# 用ward凝聚聚类提取簇
agglomerative = AgglomerativeClustering(n_clusters=40)
labels_agg = agglomerative.fit_predict(X_pca)
print("cluster sizes agglomerative clustering: {}".format(np.bincount(labels_agg)))

n_clusters = 40
for cluster in [10, 13, 19, 22, 36]: # 手动挑选“有趣的”簇
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 15, subplot_kw={'xticks': (), 'yticks': ()},
                             figsize=(15, 8))
    cluster_size = np.sum(mask)
    axes[0].set_ylabel("#{}: {}".format(cluster, cluster_size))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
    for i in range(cluster_size, 15):
        axes[i].set_visible(False)
```

Out[86]:

```
cluster sizes agglomerative clustering:
 [ 58  80  79  40 222  50  55  78 172  28  26  34  14  11  60  66 152  27
  47  31  54   5   8  56   3   5   8  18  22  82  37  89  28  24  41  40
  21  10 113  69]
```

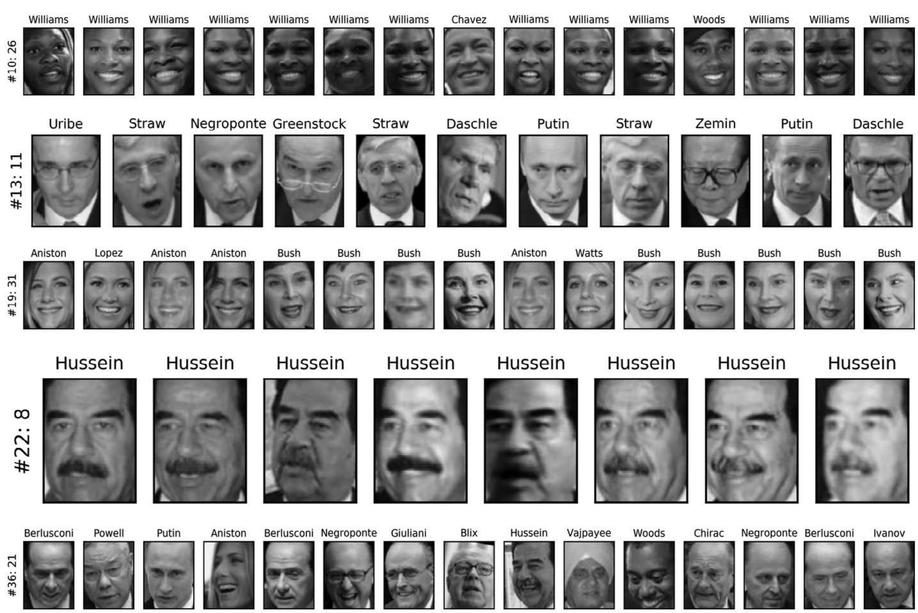


图 3-47：将簇的数量设置为 40 时，从凝聚聚类找到的簇中挑选的图像——左侧文本表示簇的编号和簇中的点的总数

这里聚类挑选出的似乎是“深色皮肤且微笑”“有领子的衬衫”“微笑的女性”“萨达姆”和“高额头”。如果进一步详细分析，我们还可以利用树状图找到这些高度相似的簇。

3.5.5 聚类方法小结

本节的内容表明，聚类的应用与评估是一个非常定性的过程，通常在数据分析的探索阶段很有帮助。我们学习了三种聚类算法：k 均值、DBSCAN 和凝聚聚类。这三种算法都可以控制聚类的粒度（granularity）。k 均值和凝聚聚类允许你指定想要的簇的数量，而 DBSCAN 允许你用 `eps` 参数定义接近程度，从而间接影响簇的大小。三种方法都可以用于大型的现实世界数据集，都相对容易理解，也都可以聚类成多个簇。

每种算法的优点稍有不同。k 均值可以用簇的平均值来表示簇。它还可以被看作一种分解方法，每个数据点都由其簇中心表示。DBSCAN 可以检测到没有分配任何簇的“噪声点”，还可以帮助自动判断簇的数量。与其他两种方法不同，它允许簇具有复杂的形状，正如我们在 `two_moons` 的例子中所看到的那样。DBSCAN 有时会生成大小差别很大的簇，这可能是它的优点，也可能是缺点。凝聚聚类可以提供数据的可能划分的整个层次结构，可以通过树状图轻松查看。

3.6 小结与展望

本章介绍了一系列无监督学习算法，可用于探索性数据分析和预处理。找到数据的正确表示对于监督学习和无监督学习的成功通常都至关重要，预处理和分解方法在数据准备中具有重要作用。

分解、流形学习和聚类都是加深数据理解的重要工具，在没有监督信息的情况下，也是理解数据的仅有的方法。即使是在监督学习中，探索性工具对于更好地理解数据性质也很重要。通常来说，很难量化无监督算法的有用性，但这不应该妨碍你使用它们来深入理解数据。学完这些方法，你就已经掌握了机器学习从业者每天使用的所有必要的学习算法。

我们建议你在 `scikit-learn` 中包含的二维玩具数据和现实世界数据集（比如 `digits`、`iris` 和 `cancer` 数据集）上尝试聚类和分解方法。

估计器接口小结

我们简要回顾一下第 2 章和第 3 章介绍的 API。scikit-learn 中的所有算法——无论是预处理、监督学习还是无监督学习算法——都被实现为类。这些类在 scikit-learn 中叫作**估计器**（estimator）。为了应用算法，你首先需要将特定类的对象实例化：

In[87]:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
```

估计器类包含算法，也保存了利用算法从数据中学到的模型。

在构建模型对象时，你应该设置模型的所有参数。这些参数包括正则化、复杂度控制、要找到的簇的数量，等等。所有估计器都有 fit 方法，用于构建模型。fit 方法要求第一个参数总是数据 X，用一个 NumPy 数组或 SciPy 稀疏矩阵表示，其中每一行代表一个数据点。数据 X 总被假定为具有连续值（浮点数）的 NumPy 数组或 SciPy 稀疏矩阵。监督算法还需要有一个 y 参数，它是一维 NumPy 数组，包含回归或分类的目标值（即已知的输出标签或响应）。

在 scikit-learn 中，应用学到的模型主要有两种方法。要想创建一个新输出形式（比如 y）的预测，可以用 predict 方法。要想创建输入数据 X 的一种新表示，可以用 transform 方法。表 3-1 汇总了 predict 方法和 transform 方法的使用场景。

表3-1：scikit-learn API小结

estimator.fit(X_train, [y_train])	
estimator.predict(X_test)	estimator.transform(X_test)
分类	预处理
回归	降维
聚类	特征提取
	特征选择

此外，所有监督模型都有 score(X_test, y_test) 方法，可以评估模型。在表 3-1 中，X_train 和 y_train 指的是训练数据和训练标签，而 X_test 和 y_test 指的是测试数据和测试标签（如果适用的话）。

第 4 章

数据表示与特征工程

到目前为止，我们一直假设数据是由浮点数组成的二维数组，其中每一列是描述数据点的**连续特征**（continuous feature）。对于许多应用而言，数据的收集方式并不是这样。一种特别常见的特征类型就是**分类特征**（categorical feature），也叫**离散特征**（discrete feature）。这种特征通常并不是数值。分类特征与连续特征之间的区别类似于分类和回归之间的区别，只是前者在输入端而不是输出端。我们已经见过的连续特征的例子包括像素明暗程度和花的尺寸测量。分类特征的例子包括产品的品牌、产品的颜色或产品的销售部门（图书、服装、硬件）。这些都是描述一件产品的属性，但它们不以连续的方式变化。一件产品要么属于服装部门，要么属于图书部门。在图书和服装之间没有中间部门，不同的分类之间也没有顺序（图书不大于服装也不小于服装，硬件不在图书和服装之间，等等）。

无论你的数据包含哪种类型的特征，数据表示方式都会对机器学习模型的性能产生巨大影响。我们在第 2 章和第 3 章中看到，数据缩放非常重要。换句话说，如果你没有缩放数据（比如，缩放到单位方差），那么你用厘米还是英寸表示测量数据的结果将会不同。我们在第 2 章中还看到，用额外的特征**扩充**（augment）数据也很有帮助，比如添加特征的交互项（乘积）或更一般的多项式。

对于某个特定应用来说，如何找到最佳数据表示，这个问题被称为**特征工程**（feature engineering），它是数据科学家和机器学习从业者在尝试解决现实世界问题时的主要任务之一。用正确的方式表示数据，对监督模型性能的影响比所选择的精确参数还要大。

本章我们将首先学习分类特征非常重要而又非常常见的例子，然后对于特征和模型的特定组合给出一些有用的变换示例。

4.1 分类变量

作为例子，我们将使用美国成年人收入的数据集，该数据集是从 1994 年的普查数据库中导

出的。`adult` 数据集的任务是预测一名工人的收入是高于 50 000 美元还是低于 50 000 美元。这个数据集的特征包括工人的年龄、雇用方式（独立经营、私营企业员工、政府职员等）、教育水平、性别、每周工作时长、职业，等等。表 4-1 给出了该数据集中的前几个条目。

表4-1：`adult`数据集的前几个条目

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K
5	37	Private	Masters	Female	40	Exec-managerial	<=50K
6	49	Private	9th	Female	16	Other-service	<=50K
7	52	Self-emp-not-inc	HS-grad	Male	45	Exec-managerial	>50K
8	31	Private	Masters	Female	50	Prof-specialty	>50K
9	42	Private	Bachelors	Male	40	Exec-managerial	>50K
10	37	Private	Some-college	Male	80	Exec-managerial	>50K

这个任务属于分类任务，两个类别是收入 `<=50k` 和 `>50k`。也可以预测具体收入，那样就变成了一个回归任务。但那样问题将变得更加困难，而理解 50K 的分界线本身也很有趣。

在这个数据集中，`age`（年龄）和 `hours-per-week`（每周工作时长）是连续特征，我们知道如何处理这种特征。但 `workclass`（工作类型）、`education`（教育程度）、`gender`（性别）、`occupation`（职业）都是分类特征。它们都来自一系列固定的可能取值（而不是一个范围），表示的是定性属性（而不是数量）。

首先，假设我们想要在这个数据上学习一个 Logistic 回归分类器。我们在第 2 章学过，Logistic 回归利用下列公式进行预测，预测值为 $\hat{y}$：

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \cdots + w[p] * x[p] + b > 0$$

其中 $w[i]$ 和 b 是从训练集中学到的系数，$x[i]$ 是输入特征。当 $x[i]$ 是数字时这个公式才有意义，但如果 $x[2]$ 是 `"Masters"` 或 `"Bachelors"` 的话，这个公式则没有意义。显然，在应用 Logistic 回归时，我们需要换一种方式来表示数据。下一节将会说明我们如何解决这一问题。

4.1.1　One-Hot编码（虚拟变量）

到目前为止，表示分类变量最常用的方法就是使用 **one-hot 编码**（one-hot-encoding）或 **N 取一编码**（one-out-of-N encoding），也叫**虚拟变量**（dummy variable）。虚拟变量背后的思想是将一个分类变量替换为一个或多个新特征，新特征取值为 0 和 1。对于线性二分类（以及 `scikit-learn` 中其他所有模型）的公式而言，0 和 1 这两个值是有意义的，我们可以像这样对每个类别引入一个新特征，从而表示任意数量的类别。

比如说，workclass 特征的可能取值包括 "Government Employee"、"Private Employee"、"Self Employed" 和 "Self Employed Incorporated"。为了编码这 4 个可能的取值，我们创建了 4 个新特征，分别叫作 "Government Employee"、"Private Employee"、"Self Employed" 和 "Self Employed Incorporated"。如果一个人的 workclass 取某个值，那么对应的特征取值为 1，其他特征均取值为 0。因此，对每个数据点来说，4 个新特征中只有一个的取值为 1。这就是它叫作 one-hot 编码或 N 取一编码的原因。

其原理如表 4-2 所示。利用 4 个新特征对一个特征进行编码。在机器学习算法中使用此数据时，我们将会删除原始的 workclass 特征，仅保留 0-1 特征。

表4-2：利用one-hot编码来编码workclass特征

workclass	Government Employee	Private Employee	Self Employed	Self Employed Incorporated
Government Employee	1	0	0	0
Private Employee	0	1	0	0
Self Employed	0	0	1	0
Self Employed Incorporated	0	0	0	1

我们使用的 one-hot 编码与统计学中使用的虚拟编码（dummy encoding）非常相似，但并不完全相同。为简单起见，我们将每个类别编码为不同的二元特征。在统计学中，通常将具有 *k* 个可能取值的分类特征编码为 *k*−1 个特征（都等于零表示最后一个可能取值）。这么做是为了简化分析（更专业的说法是，这可以避免使数据矩阵秩亏）。

将数据转换为分类变量的 one-hot 编码有两种方法：一种是使用 pandas，一种是使用 scikit-learn。在写作本书时，使用 pandas 要稍微简单一些，所以我们选择这种方法。首先，我们使用 pandas 从逗号分隔值（CSV）文件中加载数据：

In[2]:

```
import pandas as pd
from IPython.display import display

# 文件中没有包含列名称的表头，因此我们传入header=None
# 然后在"names"中显式地提供列名称
data = pd.read_csv(
    "data/adult.data", header=None, index_col=False,
    names=['age', 'workclass', 'fnlwgt', 'education',  'education-num',
           'marital-status', 'occupation', 'relationship', 'race', 'gender',
           'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
           'income'])
# 为了便于说明，我们只选了其中几列
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week',
             'occupation', 'income']]
# IPython.display可以在Jupyter notebook中输出漂亮的格式
display(data.head())
```

其结果见表 4-3。

表4-3：adult数据集的前5行

	age	workclass	education	gender	hours–per–week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K

1. 检查字符串编码的分类数据

读取完这样的数据集之后，最好先检查每一列是否包含有意义的分类数据。在处理人工（比如网站用户）输入的数据时，可能没有固定的类别，拼写和大小写也存在差异，因此可能需要预处理。举个例子，有人可能将性别填为“male”（男性），有人可能填为“man”（男人），而我们希望能用同一个类别来表示这两种输入。检查列的内容有一个好方法，就是使用 `pandas Series`（`Series` 是 `DataFrame` 中单列对应的数据类型）的 `value_counts` 函数，以显示唯一值及其出现次数：

In[3]:

```
print(data.gender.value_counts())
```

Out[3]:

```
 Male      21790
 Female    10771
Name: gender, dtype: int64
```

可以看到，在这个数据集中性别刚好有两个值：`Male` 和 `Female`，这说明数据格式已经很好，可以用 one-hot 编码来表示。在实际的应用中，你应该查看并检查所有列的值。为简洁起见，这里我们将跳过这一步。

用 `pandas` 编码数据有一种非常简单的方法，就是使用 `get_dummies` 函数。`get_dummies` 函数自动变换所有具有对象类型（比如字符串）的列或所有分类的列（这是 `pandas` 中的一个特殊概念，我们还没有讲到）：

In[4]:

```
print("Original features:\n", list(data.columns), "\n")
data_dummies = pd.get_dummies(data)
print("Features after get_dummies:\n", list(data_dummies.columns))
```

Out[4]:

```
Original features:
 ['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation',
  'income']

Features after get_dummies:
['age', 'hours-per-week', 'workclass_ ?', 'workclass_ Federal-gov',
 'workclass_ Local-gov', 'workclass_ Never-worked', 'workclass_ Private',
 'workclass_ Self-emp-inc', 'workclass_ Self-emp-not-inc',
 'workclass_ State-gov', 'workclass_ Without-pay', 'education_ 10th',
 'education_ 11th', 'education_ 12th', 'education_ 1st-4th',
```

```
 ...
'education_ Preschool', 'education_ Prof-school', 'education_ Some-college',
'gender_ Female', 'gender_ Male', 'occupation_ ?',
'occupation_ Adm-clerical', 'occupation_ Armed-Forces',
'occupation_ Craft-repair', 'occupation_ Exec-managerial',
'occupation_ Farming-fishing', 'occupation_ Handlers-cleaners',
 ...
'occupation_ Tech-support', 'occupation_ Transport-moving',
'income_ <=50K', 'income_ >50K']
```

你可以看到，连续特征 age 和 hours-per-week 没有发生变化，而分类特征的每个可能取值都被扩展为一个新特征：

In[5]:

```
data_dummies.head()
```

Out[5]:

	age	hours–per–week	workclass_?	workclass_ Federal–gov	workclass_ Local–gov	...	occupation_ Tech–support	occupation_ Transport–moving	income_ <=50K	income_ >50K
0	39	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
1	50	13	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
2	38	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
3	53	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
4	28	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0

5 rows × 46 columns

下面我们可以使用 values 属性将 data_dummies 数据框（DataFrame）转换为 NumPy 数组，然后在其上训练一个机器学习模型。在训练模型之前，注意要把目标变量（现在被编码为两个 income 列）从数据中分离出来。将输出变量或输出变量的一些导出属性包含在特征表示中，这是构建监督机器学习模型时一个非常常见的错误。

注意：pandas 中的列索引包括范围的结尾，因此 'age':'occupation_ Transport-moving' 中包括 occupation_ Transport-moving。这与 NumPy 数组的切片不同，后者不包括范围的结尾，例如 np.arange(11)[0:10] 不包括索引编号为 10 的元素。

在这个例子中，我们仅提取包含特征的列，也就是从 age 到 occupation_ Transport-moving 的所有列。这一范围包含所有特征，但不包含目标：

In[6]:

```
features = data_dummies.ix[:, 'age':'occupation_ Transport-moving']
# 提取NumPy数组
X = features.values
y = data_dummies['income_ >50K'].values
print("X.shape: {}  y.shape: {}".format(X.shape, y.shape))
```

Out[6]:

```
X.shape: (32561, 44) y.shape: (32561,)
```

现在数据的表示方式可以被 scikit-learn 处理，我们可以像之前一样继续下一步：

In[7]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Test score: {:.2f}".format(logreg.score(X_test, y_test)))
```

Out[7]:

```
Test score: 0.81
```

> 在这个例子中，我们对同时包含训练数据和测试数据的数据框调用 get_dummies。这一点很重要，可以确保训练集和测试集中分类变量的表示方式相同。
>
> 假设我们的训练集和测试集位于两个不同的数据框中。如果 workclass 特征的 "Private Employee" 取值没有出现在测试集中，那么 pandas 会认为这个特征只有 3 个可能的取值，因此只会创建 3 个新的虚拟特征。现在训练集和测试集的特征个数不相同，我们就无法将在训练集上学到的模型应用到测试集上。更糟糕的是，假设 workclass 特征在训练集中有 "Government Employee" 和 "Private Employee" 两个值，而在测试集中有 "Self Employed" 和 "Self Employed Incorporated" 两个值。在两种情况下，pandas 都会创建两个新的虚拟特征，所以编码后的数据框的特征个数相同。但在训练集和测试集中的两个虚拟特征含义完全不同。训练集中表示 "Government Employee" 的那一列在测试集中对应的是 "Self Employed"。
>
> 如果我们在这个数据上构建机器学习模型，那么它的表现会很差，因为它认为每一列表示的是相同的内容（因为位置相同），而实际上表示的却是非常不同的内容。要想解决这个问题，可以在同时包含训练数据点和测试数据点的数据框上调用 get_dummies，也可以确保调用 get_dummies 后训练集和测试集的列名称相同，以保证它们具有相同的语义。

4.1.2　数字可以编码分类变量

在 adult 数据集的例子中，分类变量被编码为字符串。一方面，可能会有拼写错误；但另一方面，它明确地将一个变量标记为分类变量。无论是为了便于存储还是因为数据的收集方式，分类变量通常被编码为整数。例如，假设 adult 数据集中的人口普查数据是利用问卷收集的，workclass 的回答被记录为 0（在第一个框打勾）、1（在第二个框打勾）、2（在第三个框打勾），等等。现在该列包含数字 0 到 8，而不是像 "Private" 这样的字符串。如果有人观察表示数据集的表格，很难一眼看出这个变量应该被视为连续变量还是分类变

量。但是，如果知道这些数字表示的是就业状况，那么很明显它们是不同的状态，不应该用单个连续变量来建模。

> 分类特征通常用整数进行编码。它们是数字并不意味着它们必须被视为连续特征。一个整数特征应该被视为连续的还是离散的（one-hot 编码的），有时并不明确。如果在被编码的语义之间没有顺序关系（比如 workclass 的例子），那么特征必须被视为离散特征。对于其他情况（比如五星评分），哪种编码更好取决于具体的任务和数据，以及使用哪种机器学习算法。

pandas 的 get_dummies 函数将所有数字看作是连续的，不会为其创建虚拟变量。为了解决这个问题，你可以使用 scikit-learn 的 OneHotEncoder，指定哪些变量是连续的、哪些变量是离散的，你也可以将数据框中的数值列转换为字符串。为了说明这一点，我们创建一个两列的 DataFrame 对象，其中一列包含字符串，另一列包含整数：

In[8]:

```
# 创建一个DataFrame，包含一个整数特征和一个分类字符串特征
demo_df = pd.DataFrame({'Integer Feature': [0, 1, 2, 1],
                        'Categorical Feature': ['socks', 'fox', 'socks', 'box']})
display(demo_df)
```

其结果见表 4-4。

表4-4：包含分类字符串特征和整数特征的数据框

	Categorical Feature	Integer Feature
0	socks	0
1	fox	1
2	socks	2
3	box	1

使用 get_dummies 只会编码字符串特征，不会改变整数特征，正如表 4-5 所示。

In[9]:

```
pd.get_dummies(demo_df)
```

表4-5：表4-4中数据的one-hot编码版本，整数特征不变

	Integer Feature	Categorical Feature_box	Categorical Feature_fox	Categorical Feature_socks
0	0	0.0	0.0	1.0
1	1	0.2	1.0	0.0
2	2	0.0	0.0	1.0
3	1	1.0	0.0	0.0

如果你想为“Integer Feature”这一列创建虚拟变量，可以使用 columns 参数显式地给出想要编码的列。于是两个特征都会被当作分类特征处理（见表 4-6）：

In[10]:

```
demo_df['Integer Feature'] = demo_df['Integer Feature'].astype(str)
pd.get_dummies(demo_df, columns=['Integer Feature', 'Categorical Feature'])
```

表4-6：对表4-4中的数据做one-hot编码，同时编码整数特征和字符串特征

	Integer Feature_0	Integer Feature_1	Integer Feature_2	Categorical Feature_box	Categorical Feature_fox	Categorical Feature_socks
0	1.0	0.0	0.0	0.0	0.0	1.0
1	0.0	1.0	0.0	0.0	1.0	0.0
2	0.0	0.0	1.0	0.0	0.0	1.0
3	0.0	1.0	0.0	1.0	0.0	0.0

4.2 分箱、离散化、线性模型与树

数据表示的最佳方法不仅取决于数据的语义，还取决于所使用的模型种类。线性模型与基于树的模型（比如决策树、梯度提升树和随机森林）是两种成员很多同时又非常常用的模型，它们在处理不同的特征表示时就具有非常不同的性质。我们回到第 2 章用过的 wave 回归数据集。它只有一个输入特征。下面是线性回归模型与决策树回归在这个数据集上的对比（见图 4-1）：

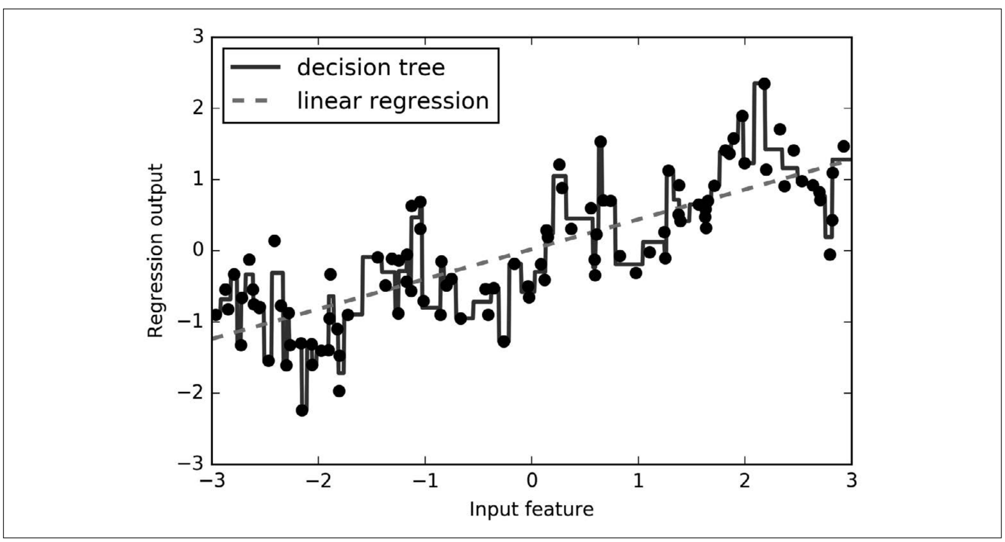


图 4-1：在 wave 数据集上比较线性回归和决策树

In[11]:

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

X, y = mglearn.datasets.make_wave(n_samples=100)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)
```

```
reg = DecisionTreeRegressor(min_samples_split=3).fit(X, y)
plt.plot(line, reg.predict(line), label="decision tree")

reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), label="linear regression")

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

正如你所知，线性模型只能对线性关系建模，对于单个特征的情况就是直线。决策树可以构建更为复杂的数据模型，但这强烈依赖于数据表示。有一种方法可以让线性模型在连续数据上变得更加强大，就是使用特征**分箱**（binning，也叫**离散化**，即 discretization）将其划分为多个特征，如下所述。

我们假设将特征的输入范围（在这个例子中是从 –3 到 3）划分成固定个数的**箱子**（bin），比如 10 个，那么数据点就可以用它所在的箱子来表示。为了确定这一点，我们首先需要定义箱子。在这个例子中，我们在 –3 和 3 之间定义 10 个均匀分布的箱子。我们用 `np.linspace` 函数创建 11 个元素，从而创建 10 个箱子，即两个连续边界之间的空间：

In[12]:

```
bins = np.linspace(-3, 3, 11)
print("bins: {}".format(bins))
```

Out[12]:

```
bins: [-3.  -2.4 -1.8 -1.2 -0.6  0.   0.6  1.2  1.8  2.4  3. ]
```

这里第一个箱子包含特征取值在 −3 到 −2.4 之间的所有数据点，第二个箱子包含特征取值在 −2.4 到 −1.8 之间的所有数据点，以此类推。

接下来，我们记录每个数据点所属的箱子。这可以用 `np.digitize` 函数轻松计算出来：

In[13]:

```
which_bin = np.digitize(X, bins=bins)
print("\nData points:\n", X[:5])
print("\nBin membership for data points:\n", which_bin[:5])
```

Out[13]:

```
Data points:
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
[-2.064]]

Bin membership for data points:
[[ 4]
 [10]
 [ 8]
 [ 6]
[ 2]]
```

我们在这里做的是将 wave 数据集中单个连续输入特征变换为一个分类特征，用于表示数据点所在的箱子。要想在这个数据上使用 scikit-learn 模型，我们利用 preprocessing 模块的 OneHotEncoder 将这个离散特征变换为 one-hot 编码。OneHotEncoder 实现的编码与 pandas.get_dummies 相同，但目前它只适用于值为整数的分类变量：

In[14]:

```
from sklearn.preprocessing import OneHotEncoder
# 使用OneHotEncoder进行变换
encoder = OneHotEncoder(sparse=False)
# encoder.fit找到which_bin中的唯一值
encoder.fit(which_bin)
# transform创建one-hot编码
X_binned = encoder.transform(which_bin)
print(X_binned[:5])
```

Out[14]:

```
[[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

由于我们指定了 10 个箱子，所以变换后的 X_binned 数据集现在包含 10 个特征：

In[15]:

```
print("X_binned.shape: {}".format(X_binned.shape))
```

Out[15]:

```
X_binned.shape: (100, 10)
```

下面我们在 one-hot 编码后的数据上构建新的线性模型和新的决策树模型。结果见图 4-2，箱子的边界由黑色虚线表示：

In[16]:

```
line_binned = encoder.transform(np.digitize(line, bins=bins))

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='linear regression binned')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='decision tree binned')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(bins, -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
```

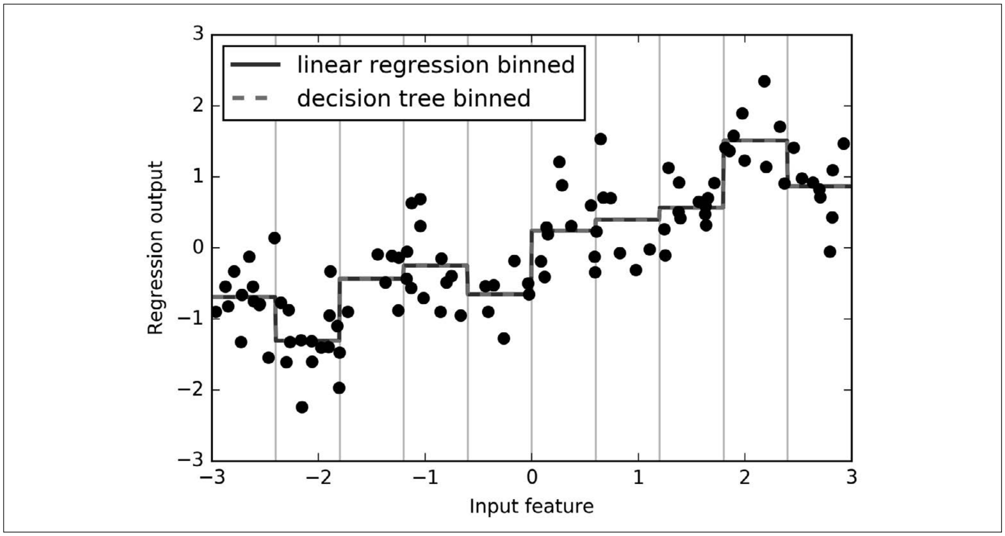


图 4-2：在分箱特征上比较线性回归和决策树回归

虚线和实线完全重合，说明线性回归模型和决策树做出了完全相同的预测。对于每个箱子，二者都预测一个常数值。因为每个箱子内的特征是不变的，所以对于一个箱子内的所有点，任何模型都会预测相同的值。比较对特征进行分箱前后模型学到的内容，我们发现，线性模型变得更加灵活了，因为现在它对每个箱子具有不同的取值，而决策树模型的灵活性降低了。分箱特征对基于树的模型通常不会产生更好的效果，因为这种模型可以学习在任何位置划分数据。从某种意义上来看，决策树可以学习如何分箱对预测这些数据最为有用。此外，决策树可以同时查看多个特征，而分箱通常针对的是单个特征。不过，线性模型的表现力在数据变换后得到了极大的提高。

对于特定的数据集，如果有充分的理由使用线性模型——比如数据集很大、维度很高，但有些特征与输出的关系是非线性的——那么分箱是提高建模能力的好方法。

4.3 交互特征与多项式特征

想要丰富特征表示，特别是对于线性模型而言，另一种方法是添加原始数据的**交互特征**（interaction feature）和**多项式特征**（polynomial feature）。这种特征工程通常用于统计建模，但也常用于许多实际的机器学习应用中。

作为第一个例子，我们再看一次图 4-2。线性模型对 wave 数据集中的每个箱子都学到一个常数值。但我们知道，线性模型不仅可以学习偏移，还可以学习斜率。想要向分箱数据上的线性模型添加斜率，一种方法是重新加入原始特征（图中的 x 轴）。这样会得到 11 维的数据集，如图 4-3 所示。

In[17]:

```
X_combined = np.hstack([X, X_binned])
print(X_combined.shape)
```

Out[17]:

```
(100, 11)
```

In[18]:

```
reg = LinearRegression().fit(X_combined, y)

line_combined = np.hstack([line, line_binned])
plt.plot(line, reg.predict(line_combined), label='linear regression combined')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')

plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.plot(X[:, 0], y, 'o', c='k')
```

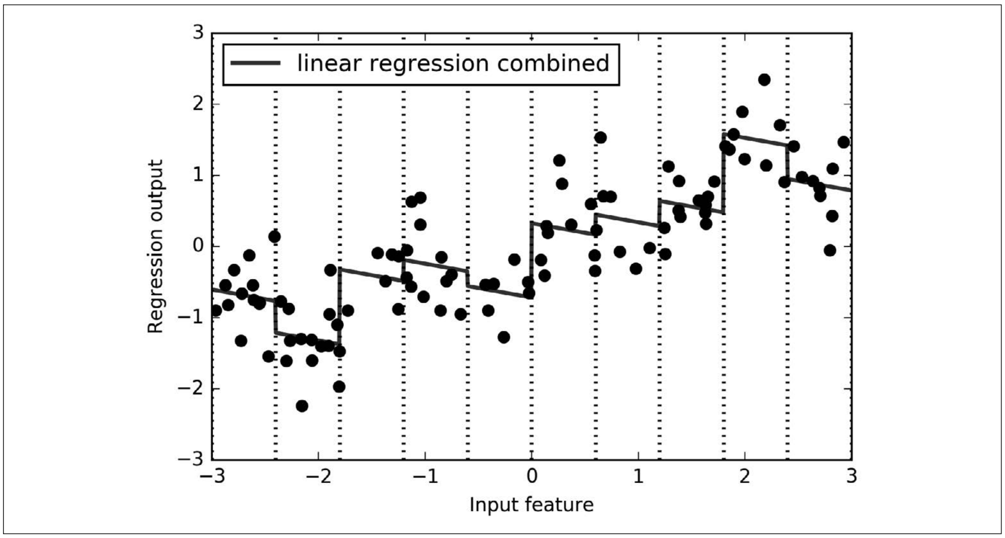


图 4-3：使用分箱特征和单一全局斜率的线性回归

在这个例子中，模型在每个箱子中都学到一个偏移，还学到一个斜率。学到的斜率是向下的，并且在所有箱子中都相同——只有一个 x 轴特征，也就只有一个斜率。因为斜率在所有箱子中是相同的，所以它似乎不是很有用。我们更希望每个箱子都有一个不同的斜率！为了实现这一点，我们可以添加交互特征或乘积特征，用来表示数据点所在的箱子以及数据点在 x 轴上的位置。这个特征是箱子指示符与原始特征的乘积。我们来创建数据集：

In[19]:

```
X_product = np.hstack([X_binned, X * X_binned])
print(X_product.shape)
```

Out[19]:

```
(100, 20)
```

这个数据集现在有 20 个特征：数据点所在箱子的指示符与原始特征和箱子指示符的乘积。你可以将乘积特征看作每个箱子 x 轴特征的单独副本。它在箱子内等于原始特征，在其他位置等于零。图 4-4 给出了线性模型在这种新表示上的结果：

In[20]:

```
reg = LinearRegression().fit(X_product, y)

line_product = np.hstack([line_binned, line * line_binned])
plt.plot(line, reg.predict(line_product), label='linear regression product')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

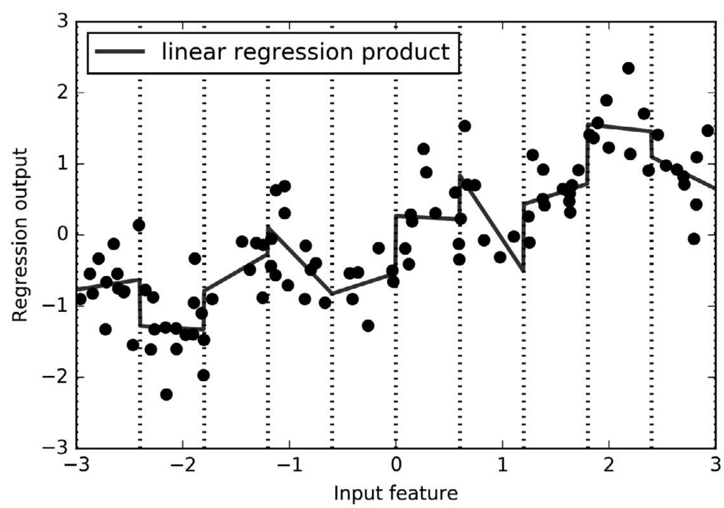


图 4-4：每个箱子具有不同斜率的线性回归

如你所见，现在这个模型中每个箱子都有自己的偏移和斜率。

使用分箱是扩展连续特征的一种方法。另一种方法是使用原始特征的**多项式**（polynomial）。对于给定特征 x，我们可以考虑 `x ** 2`、`x ** 3`、`x ** 4`，等等。这在 `preprocessing` 模块的 `PolynomialFeatures` 中实现：

In[21]:

```
from sklearn.preprocessing import PolynomialFeatures

# 包含直到x ** 10的多项式:
# 默认的"include_bias=True"添加恒等于1的常数特征
poly = PolynomialFeatures(degree=10, include_bias=False)
```

```
poly.fit(X)
X_poly = poly.transform(X)
```

多项式的次数为 10，因此生成了 10 个特征：

In[22]:

```
print("X_poly.shape: {}".format(X_poly.shape))
```

Out[22]:

```
X_poly.shape: (100, 10)
```

我们比较 X_poly 和 X 的元素：

In[23]:

```
print("Entries of X:\n{}".format(X[:5]))
print("Entries of X_poly:\n{}".format(X_poly[:5]))
```

Out[23]:

```
Entries of X:
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]
Entries of X_poly:
[[    -0.753        0.567       -0.427        0.321       -0.242      0.182
      -0.137        0.103       -0.078        0.058]
 [     2.704        7.313       19.777       53.482      144.632    391.125
    1057.714     2860.360     7735.232    20918.278]
 [     1.392        1.938        2.697        3.754        5.226      7.274
      10.125       14.094       19.618       27.307]
 [     0.592        0.350        0.207        0.123        0.073      0.043
       0.025        0.015        0.009        0.005]
 [    -2.064        4.260       -8.791       18.144      -37.448     77.289
    -159.516      329.222     -679.478     1402.367]]
```

你可以通过调用 get_feature_names 方法来获取特征的语义，给出每个特征的指数：

In[24]:

```
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

Out[24]:

```
Polynomial feature names:
['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9', 'x0^10']
```

你可以看到，X_poly 的第一列与 X 完全对应，而其他列则是第一列的幂。有趣的是，你可以发现有些值非常大。第二行有大于 20 000 的元素，数量级与其他行都不相同。

将多项式特征与线性回归模型一起使用，可以得到经典的**多项式回归**（polynomial regression）模型（见图 4-5）：

In[26]:

```
reg = LinearRegression().fit(X_poly, y)
```

```
line_poly = poly.transform(line)
plt.plot(line, reg.predict(line_poly), label='polynomial linear regression')
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

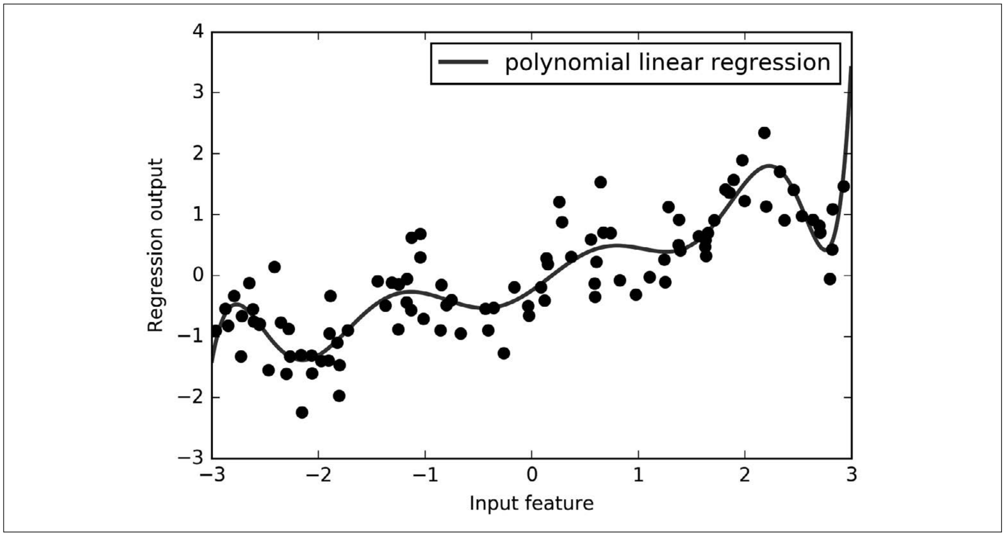


图 4-5：具有 10 次多项式特征的线性回归

如你所见，多项式特征在这个一维数据上得到了非常平滑的拟合。但高次多项式在边界上或数据很少的区域可能有极端的表现。

作为对比，下面是在原始数据上学到的核 SVM 模型，没有做任何变换（见图 4-6）：

In[26]:

```
from sklearn.svm import SVR

for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line), label='SVR gamma={}'.format(gamma))

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

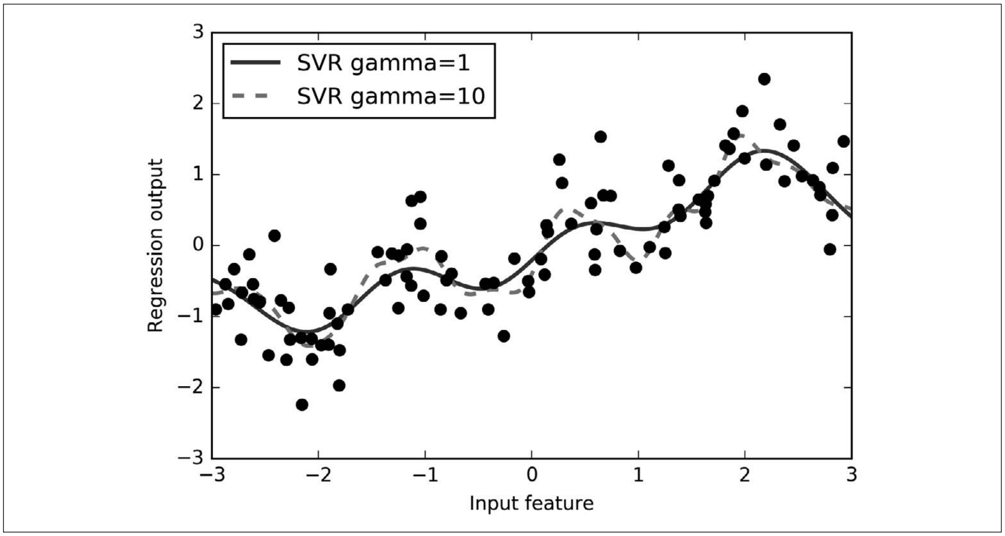


图 4-6：对于 RBF 核的 SVM，使用不同 gamma 参数的对比

使用更加复杂的模型（即核 SVM），我们能够学到一个与多项式回归的复杂度类似的预测结果，且不需要进行显式的特征变换。

我们再次观察波士顿房价数据集，作为对交互特征和多项式特征更加实际的应用。我们在第 2 章已经在这个数据集上使用过多项式特征了。现在来看一下这些特征的构造方式，以及多项式特征的帮助有多大。首先加载数据，然后利用 MinMaxScaler 将其缩放到 0 和 1 之间：

In[27]:

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split\
    (boston.data, boston.target, random_state=0)

# 缩放数据
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

下面我们提取多项式特征和交互特征，次数最高为 2：

In[28]:

```
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_poly.shape: {}".format(X_train_poly.shape))
```

Out[28]:

```
X_train.shape: (379, 13)
X_train_poly.shape: (379, 105)
```

原始数据有 13 个特征，现在被扩展到 105 个交互特征。这些新特征表示两个不同的原始特征之间所有可能的交互项，以及每个原始特征的平方。这里 degree=2 的意思是，我们需要由最多两个原始特征的乘积组成的所有特征。利用 get_feature_names 方法可以得到输入特征和输出特征之间的确切对应关系：

In[29]:

```
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

Out[29]:

```
Polynomial feature names:
['1', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10',
 'x11', 'x12', 'x0^2', 'x0 x1', 'x0 x2', 'x0 x3', 'x0 x4', 'x0 x5', 'x0 x6',
 'x0 x7', 'x0 x8', 'x0 x9', 'x0 x10', 'x0 x11', 'x0 x12', 'x1^2', 'x1 x2',
 'x1 x3', 'x1 x4', 'x1 x5', 'x1 x6', 'x1 x7', 'x1 x8', 'x1 x9', 'x1 x10',
 'x1 x11', 'x1 x12', 'x2^2', 'x2 x3', 'x2 x4', 'x2 x5', 'x2 x6', 'x2 x7',
 'x2 x8', 'x2 x9', 'x2 x10', 'x2 x11', 'x2 x12', 'x3^2', 'x3 x4', 'x3 x5',
 'x3 x6', 'x3 x7', 'x3 x8', 'x3 x9', 'x3 x10', 'x3 x11', 'x3 x12', 'x4^2',
 'x4 x5', 'x4 x6', 'x4 x7', 'x4 x8', 'x4 x9', 'x4 x10', 'x4 x11', 'x4 x12',
 'x5^2', 'x5 x6', 'x5 x7', 'x5 x8', 'x5 x9', 'x5 x10', 'x5 x11', 'x5 x12',
 'x6^2', 'x6 x7', 'x6 x8', 'x6 x9', 'x6 x10', 'x6 x11', 'x6 x12', 'x7^2',
 'x7 x8', 'x7 x9', 'x7 x10', 'x7 x11', 'x7 x12', 'x8^2', 'x8 x9', 'x8 x10',
 'x8 x11', 'x8 x12', 'x9^2', 'x9 x10', 'x9 x11', 'x9 x12', 'x10^2', 'x10 x11',
 'x10 x12', 'x11^2', 'x11 x12', 'x12^2']
```

第一个新特征是常数特征，这里的名称是 "1"。接下来的 13 个特征是原始特征（名称从 "x0" 到 "x12"）。然后是第一个特征的平方（"x0^2"）以及它与其他特征的组合。

我们对 Ridge 在有交互特征的数据上和没有交互特征的数据上的性能进行对比：

In[30]:

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(
    ridge.score(X_test_scaled, y_test)))
ridge = Ridge().fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(
    ridge.score(X_test_poly, y_test)))
```

Out[30]:

```
Score without interactions: 0.621
Score with interactions: 0.753
```

显然，在使用 Ridge 时，交互特征和多项式特征对性能有很大的提升。但如果使用更加复杂的模型（比如随机森林），情况会稍有不同：

In[31]:

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100).fit(X_train_scaled, y_train)
```

```
print("Score without interactions: {:.3f}".format(
    rf.score(X_test_scaled, y_test)))
rf = RandomForestRegressor(n_estimators=100).fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(rf.score(X_test_poly, y_test)))
```

Out[31]:

```
Score without interactions: 0.799
Score with interactions: 0.763
```

你可以看到，即使没有额外的特征，随机森林的性能也要优于 Ridge。添加交互特征和多项式特征实际上会略微降低其性能。

4.4　单变量非线性变换

我们刚刚看到，添加特征的平方或立方可以改进线性回归模型。其他变换通常也对变换某些特征有用，特别是应用数学函数，比如 log、exp 或 sin。虽然基于树的模型只关注特征的顺序，但线性模型和神经网络依赖于每个特征的尺度和分布。如果在特征和目标之间存在非线性关系，那么建模就变得非常困难，特别是对于回归问题。log 和 exp 函数可以帮助调节数据的相对比例，从而改进线性模型或神经网络的学习效果。我们在第 2 章中对内存价格数据应用过这种函数。在处理具有周期性模式的数据时，sin 和 cos 函数非常有用。

大部分模型都在每个特征（在回归问题中还包括目标值）大致遵循高斯分布时表现最好，也就是说，每个特征的直方图应该具有类似于熟悉的“钟形曲线”的形状。使用诸如 log 和 exp 之类的变换并不稀奇，但却是实现这一点的简单又有效的方法。在一种特别常见的情况下，这样的变换非常有用，就是处理整数计数数据时。计数数据是指类似“用户 A 多长时间登录一次？”这样的特征。计数不可能取负值，并且通常遵循特定的统计模式。下面我们使用一个模拟的计数数据集，其性质与在自然状态下能找到的数据集类似。特征全都是整数值，而响应是连续的：

In[32]:

```
rnd = np.random.RandomState(0)
X_org = rnd.normal(size=(1000, 3))
w = rnd.normal(size=3)

X = rnd.poisson(10 * np.exp(X_org))
y = np.dot(X_org, w)
```

我们来看一下第一个特征的前 10 个元素。它们都是正整数，但除此之外很难找出特定的模式。

如果我们计算每个值的出现次数，那么数值的分布将变得更清楚：

In[33]:

```
print("Number of feature appearances:\n{}".format(np.bincount(X[:, 0])))
```

Out[33]:

```
Number of feature appearances:
[28 38 68 48 61 59 45 56 37 40 35 34 36 26 23 26 27 21 23 23 18 21 10  9 17
  9  7 14 12  7  3  8  4  5  5  3  4  2  4  1  1  3  2  5  3  8  2  5  2  1
```

```
2  3  3  2  2  3  3  0  1  2  1  0  0  3  1  0  0  0  1  3  0  1  0  2  0
1  1  0  0  0  0  1  0  0  2  2  0  1  1  0  0  0  0  1  1  0  0  0  0  0
0  0  1  0  0  0  0  0  1  1  0  0  1  0  0  0  0  0  0  0  1  0  0  0  0
1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1]
```

数字 2 似乎是最常见的，共出现了 68 次（bincount 始终从 0 开始），更大数字的出现次数快速下降。但也有一些很大的数字，比如 134 出现了 2 次[1]。我们在图 4-7 中将计数可视化。

In[34]:

```
bins = np.bincount(X[:, 0])
plt.bar(range(len(bins)), bins, color='w')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```

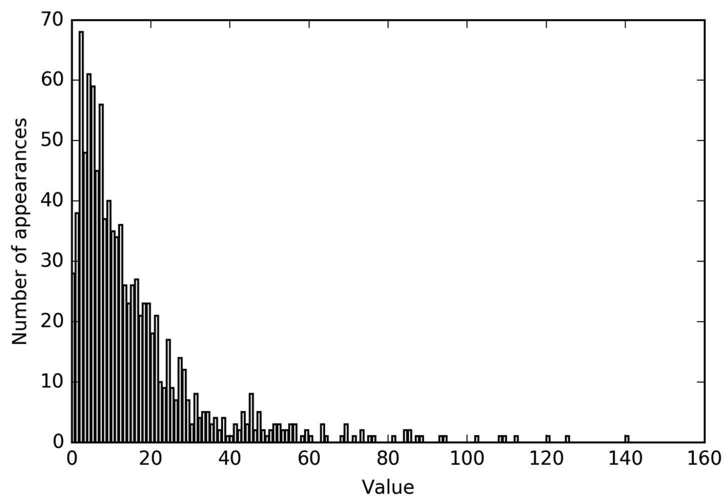


图 4-7：X[:, 0] 特征取值的直方图

特征 X[:, 1] 和 X[:, 2] 具有类似的性质。这种类型的数值分布（许多较小的值和一些非常大的值）在实践中非常常见。[2] 但大多数线性模型无法很好地处理这种数据。我们尝试拟合一个岭回归模型：

In[35]:

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
score = Ridge().fit(X_train, y_train).score(X_test, y_test)
print("Test score: {:.3f}".format(score))
```

Out[35]:

```
Test score: 0.622
```

注 1：这里 134 实际的出现次数是 0，但 84 和 85 的出现次数是 2，作者想要表达的意思是没错的。——译者注

注 2：这是泊松分布，对计数数据相当重要。

你可以从相对较小的 R^2 分数中看出，Ridge 无法真正捕捉到 X 和 y 之间的关系。不过应用对数变换可能有用。由于数据取值中包括 0（对数在 0 处没有定义），所以我们不能直接应用 log，而是要计算 log(X + 1)：

In[36]:

```
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

变换之后，数据分布的不对称性变小，也不再有非常大的异常值（见图 4-8）：

In[37]:

```
plt.hist(X_train_log[:, 0], bins=25, color='gray')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```

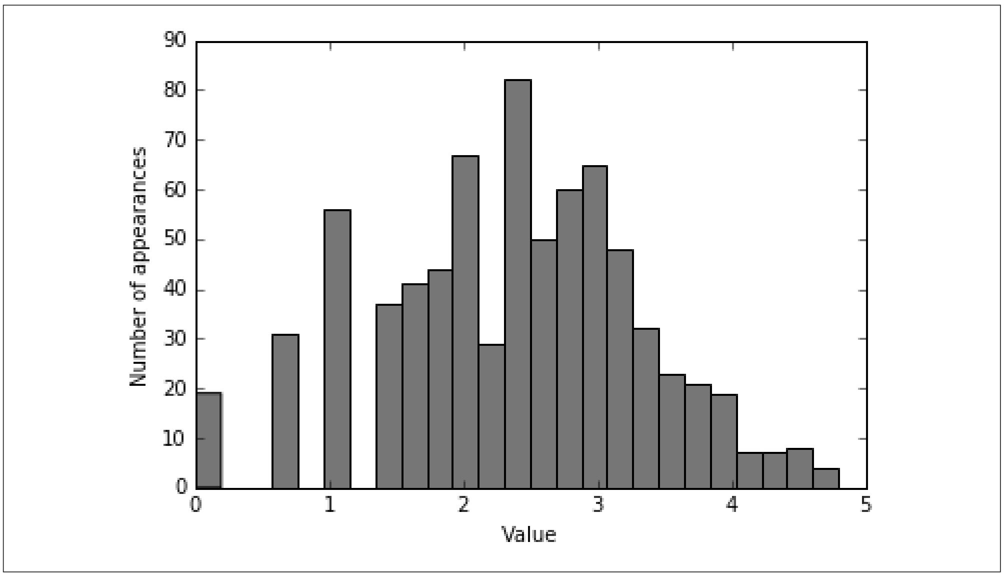


图 4-8：对 X[:, 0] 特征取值进行对数变换后的直方图

在新数据上构建一个岭回归模型，可以得到更好的拟合：

In[38]:

```
score = Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
print("Test score: {:.3f}".format(score))
```

Out[38]:

```
Test score: 0.875
```

为数据集和模型的所有组合寻找最佳变换，这在某种程度上是一门艺术。在这个例子中，所有特征都具有相同的性质，这在实践中是非常少见的情况。通常来说，只有一部分特征应该进行变换，有时每个特征的变换方式也各不相同。前面提到过，对基于树的模型而言，这种变换并不重要，但对线性模型来说可能至关重要。对回归的目标变量 y 进行变换

有时也是一个好主意。尝试预测计数（比如订单数量）是一项相当常见的任务，而且使用 `log(y + 1)` 变换也往往有用。[3]

从前面的例子中可以看出，分箱、多项式和交互项都对模型在给定数据集上的性能有很大影响，对于复杂度较低的模型更是这样，比如线性模型和朴素贝叶斯模型。与之相反，基于树的模型通常能够自己发现重要的交互项，大多数情况下不需要显式地变换数据。其他模型，比如 SVM、最近邻和神经网络，有时可能会从使用分箱、交互项或多项式中受益，但其效果通常不如线性模型那么明显。

4.5　自动化特征选择

有了这么多种创建新特征的方法，你可能会想要增大数据的维度，使其远大于原始特征的数量。但是，添加更多特征会使所有模型变得更加复杂，从而增大过拟合的可能性。在添加新特征或处理一般的高维数据集时，最好将特征的数量减少到只包含最有用的那些特征，并删除其余特征。这样会得到泛化能力更好、更简单的模型。但你如何判断每个特征的作用有多大呢？有三种基本的策略：**单变量统计**（univariate statistics）、**基于模型的选择**（model-based selection）和**迭代选择**（iterative selection）。我们将详细讨论这三种策略。所有这些方法都是监督方法，即它们需要目标值来拟合模型。这也就是说，我们需要将数据划分为训练集和测试集，并只在训练集上拟合特征选择。

4.5.1　单变量统计

在单变量统计中，我们计算每个特征和目标值之间的关系是否存在统计显著性，然后选择具有最高置信度的特征。对于分类问题，这也被称为**方差分析**（analysis of variance，ANOVA）。这些测试的一个关键性质就是它们是**单变量的**（univariate），即它们只单独考虑每个特征。因此，如果一个特征只有在与另一个特征合并时才具有信息量，那么这个特征将被舍弃。单变量测试的计算速度通常很快，并且不需要构建模型。另一方面，它们完全独立于你可能想要在特征选择之后应用的模型。

想要在 `scikit-learn` 中使用单变量特征选择，你需要选择一项测试——对分类问题通常是 `f_classif`（默认值），对回归问题通常是 `f_regression`——然后基于测试中确定的 p 值来选择一种舍弃特征的方法。所有舍弃参数的方法都使用阈值来舍弃所有 p 值过大的特征（意味着它们不可能与目标值相关）。计算阈值的方法各有不同，最简单的是 `SelectKBest` 和 `SelectPercentile`，前者选择固定数量的 k 个特征，后者选择固定百分比的特征。我们将分类的特征选择应用于 `cancer` 数据集。为了使任务更难一点，我们将向数据中添加一些没有信息量的噪声特征。我们期望特征选择能能够识别没有信息量的特征并删除它们：

In[39]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile
from sklearn.model_selection import train_test_split
```

注 3：这是对泊松分布非常粗略的近似，而从概率的角度来看，这是正确的解决方法。

```
cancer = load_breast_cancer()

# 获得确定性的随机数
rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
# 向数据中添加噪声特征
# 前30个特征来自数据集，后50个是噪声
X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(
    X_w_noise, cancer.target, random_state=0, test_size=.5)
# 使用f_classif（默认值）和SelectPercentile来选择50%的特征
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
# 对训练集进行变换
X_train_selected = select.transform(X_train)

print("X_train.shape: {}".format(X_train.shape))
print("X_train_selected.shape: {}".format(X_train_selected.shape))
```

Out[39]:

```
X_train.shape: (284, 80)
X_train_selected.shape: (284, 40)
```

如你所见，特征的数量从 80 减少到 40（原始特征数量的 50%）。我们可以用 `get_support` 方法来查看哪些特征被选中，它会返回所选特征的布尔遮罩（mask）（其可视化见图 4-9）：

In[40]:

```
mask = select.get_support()
print(mask)
# 将遮罩可视化——黑色为True，白色为False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

Out[40]:

```
[ True  True  True  True  True  True  True  True  True False  True False
  True  True  True  True  True  True False False  True  True  True  True
  True  True  True  True  True  True False False False  True False  True
  False False  True False False False False  True False False  True False
  False  True False  True False False False False False False  True False
   True False False False False  True False  True False False False False
   True  True False  True False False False False]
```

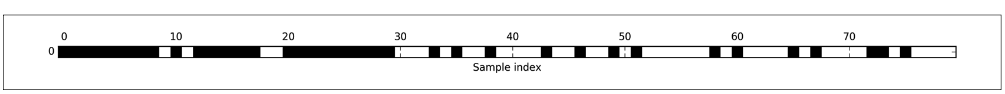


图 4-9：`SelectPercentile` 选择的特征

你可以从遮罩的可视化中看出，大多数所选择的特征都是原始特征，并且大多数噪声特征都已被删除。但原始特征的还原并不完美。我们来比较 Logistic 回归在所有特征上的性能与仅使用所选特征的性能：

In[41]:

```
from sklearn.linear_model import LogisticRegression

# 对测试数据进行变换
X_test_selected = select.transform(X_test)

lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Score with all features: {:.3f}".format(lr.score(X_test, y_test)))
lr.fit(X_train_selected, y_train)
print("Score with only selected features: {:.3f}".format(
    lr.score(X_test_selected, y_test)))
```

Out[41]:

```
Score with all features: 0.930
Score with only selected features: 0.940
```

在这个例子中，删除噪声特征可以提高性能，即使丢失了某些原始特征。这是一个非常简单的假想示例，在真实数据上的结果要更加复杂。不过，如果特征量太大以至于无法构建模型，或者你怀疑许多特征完全没有信息量，那么单变量特征选择还是非常有用的。

4.5.2　基于模型的特征选择

基于模型的特征选择使用一个监督机器学习模型来判断每个特征的重要性，并且仅保留最重要的特征。用于特征选择的监督模型不需要与用于最终监督建模的模型相同。特征选择模型需要为每个特征提供某种重要性度量，以便用这个度量对特征进行排序。决策树和基于决策树的模型提供了 feature_importances_ 属性，可以直接编码每个特征的重要性。线性模型系数的绝对值也可以用于表示特征重要性。正如我们在第 2 章所见，L1 惩罚的线性模型学到的是稀疏系数，它只用到了特征的一个很小的子集。这可以被视为模型本身的一种特征选择形式，但也可以用作另一个模型选择特征的预处理步骤。与单变量选择不同，基于模型的选择同时考虑所有特征，因此可以获取交互项（如果模型能够获取它们的话）。要想使用基于模型的特征选择，我们需要使用 SelectFromModel 变换器：

In[42]:

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold="median")
```

SelectFromModel 类选出重要性度量（由监督模型提供）大于给定阈值的所有特征。为了得到可以与单变量特征选择进行对比的结果，我们使用中位数作为阈值，这样就可以选择一半特征。我们用包含 100 棵树的随机森林分类器来计算特征重要性。这是一个相当复杂的模型，也比单变量测试要强大得多。下面我们来实际拟合模型：

In[43]:

```
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_l1.shape: {}".format(X_train_l1.shape))
```

Out[43]:

```
X_train.shape: (284, 80)
X_train_l1.shape: (284, 40)
```

我们可以再次查看选中的特征（见图 4-10）：

In[44]:

```
mask = select.get_support()
# 将遮罩可视化——黑色为True，白色为False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

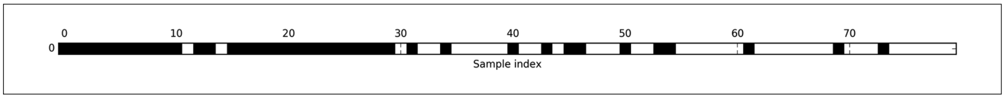


图 4-10：使用 RandomForestClassifier 的 SelectFromModel 选择的特征

这次，除了两个原始特征，其他原始特征都被选中。由于我们指定选择 40 个特征，所以也选择了一些噪声特征。我们来看一下其性能：

In[45]:

```
X_test_l1 = select.transform(X_test)
score = LogisticRegression().fit(X_train_l1, y_train).score(X_test_l1, y_test)
print("Test score: {:.3f}".format(score))
```

Out[45]:

```
Test score: 0.951
```

利用更好的特征选择，性能也得到了提高。

4.5.3 迭代特征选择

在单变量测试中，我们没有使用模型，而在基于模型的选择中，我们使用了单个模型来选择特征。在迭代特征选择中，将会构建一系列模型，每个模型都使用不同数量的特征。有两种基本方法：开始时没有特征，然后逐个添加特征，直到满足某个终止条件；或者从所有特征开始，然后逐个删除特征，直到满足某个终止条件。由于构建了一系列模型，所以这些方法的计算成本要比前面讨论过的方法更高。其中一种特殊方法是**递归特征消除**（recursive feature elimination，RFE），它从所有特征开始构建模型，并根据模型舍弃最不重要的特征，然后使用除被舍弃特征之外的所有特征来构建一个新模型，如此继续，直到仅剩下预设数量的特征。为了让这种方法能够运行，用于选择的模型需要提供某种确定特征重要性的方法，正如基于模型的选择所做的那样。下面我们使用之前用过的同一个随机森林模型，得到的结果如图 4-11 所示：

In[46]:

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
             n_features_to_select=40)
```

```
select.fit(X_train, y_train)
# 将选中的特征可视化：
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

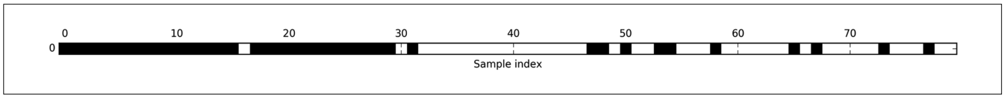


图 4-11：使用随机森林分类器模型的递归特征消除选择的特征

与单变量选择和基于模型的选择相比，迭代特征选择的结果更好，但仍然漏掉了一个特征。运行上述代码需要的时间也比基于模型的选择长得多，因为对一个随机森林模型训练了 40 次，每运行一次删除一个特征。我们来测试一下使用 RFE 做特征选择时 Logistic 回归模型的精度：

In[47]:

```
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)

score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("Test score: {:.3f}".format(score))
```

Out[47]:

```
Test score: 0.951
```

我们还可以利用在 RFE 内使用的模型来进行预测。这仅使用被选中的特征集：

In[48]:

```
print("Test score: {:.3f}".format(select.score(X_test, y_test)))
```

Out[48]:

```
Test score: 0.951
```

这里，在 RFE 内部使用的随机森林的性能，与在所选特征上训练一个 Logistic 回归模型得到的性能相同。换句话说，只要我们选择了正确的特征，线性模型的表现就与随机森林一样好。

如果你不确定何时选择使用哪些特征作为机器学习算法的输入，那么自动化特征选择可能特别有用。它还有助于减少所需要的特征数量，加快预测速度，或允许可解释性更强的模型。在大多数现实情况下，使用特征选择不太可能大幅提升性能，但它仍是特征工程工具箱中一个非常有价值的工具。

4.6 利用专家知识

对于特定应用来说，在特征工程中通常可以利用**专家知识**（expert knowledge）。虽然在许多情况下，机器学习的目的是避免创建一组专家设计的规则，但这并不意味着应该舍弃该应用或该领域的先验知识。通常来说，领域专家可以帮助找出有用的特征，其信息量比数

据原始表示要大得多。想象一下，你在一家旅行社工作，想要预测机票价格。假设你有价格以及日期、航空公司、出发地和目的地的记录。机器学习模型可能从这些记录中构建一个相当不错的模型，但可能无法学到机票价格中的某些重要因素。例如，在度假高峰月份和假日期间，机票价格通常更高。虽然某些假日的日期是固定的（比如圣诞节），其影响可以从日期中学到，但其他假日的日期可能取决于月相（比如光明节和复活节），或者由官方规定（比如学校放假）。如果每个航班都只使用公历记录日期，则无法从数据中学到这些事件。但添加一个特征是很简单的，其中编码了一个航班在公休假日或学校假期的之前、之中还是之后。利用这种方法可以将关于任务属性的先验知识编码到特征中，以辅助机器学习算法。添加一个特征并不会强制机器学习算法使用它，即使最终发现假日信息不包含关于机票价格的信息，用这一信息来扩充数据也不会有什么害处。

下面我们来看一个利用专家知识的特例——虽然在这个例子中，对这些专家知识更正确的叫法应该是“常识”。任务是预测在 Andreas 家门口的自行车出租。

在纽约，Citi Bike 运营着一个带有付费系统的自行车租赁站网络。这些站点遍布整个城市，提供了一种方便的交通方式。自行车出租数据以匿名形式公开（https://www.citibikenyc.com/system-data），并用各种方法进行了分析。我们想要解决的任务是，对于给定的日期和时间，预测有多少人将会在 Andreas 的家门口租一辆自行车——这样他就知道是否还有自行车留给他。

我们首先将这个站点 2015 年 8 月的数据加载为一个 `pandas` 数据框。我们将数据重新采样为每 3 小时一个数据，以得到每一天的主要趋势：

In[49]:

```
citibike = mglearn.datasets.load_citibike()
```

In[50]:

```
print("Citi Bike data:\n{}".format(citibike.head()))
```

Out[50]:

```
Citi Bike data:
starttime
2015-08-01 00:00:00     3.0
2015-08-01 03:00:00     0.0
2015-08-01 06:00:00     9.0
2015-08-01 09:00:00    41.0
2015-08-01 12:00:00    39.0
Freq: 3H, Name: one, dtype: float64
```

下面这个示例给出了整个月租车数量的可视化（图 4-12）：

In[51]:

```
plt.figure(figsize=(10, 3))
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(),
                       freq='D')
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")
plt.plot(citibike, linewidth=1)
plt.xlabel("Date")
plt.ylabel("Rentals")
```

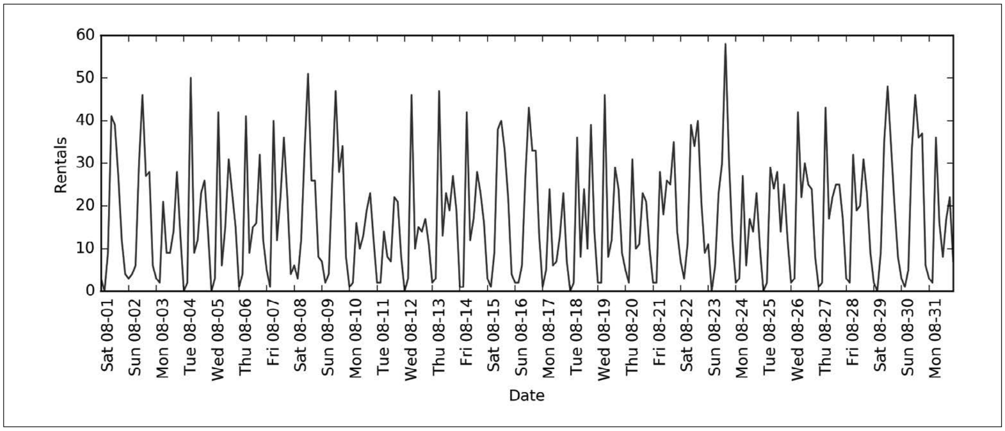


图 4-12：对于选定的 Citi Bike 站点，自行车出租数量随时间的变化

观察此数据，我们可以清楚地区分每 24 小时中的白天和夜间。工作日和周末的模式似乎也有很大不同。在对这种时间序列上的预测任务进行评估时，我们通常希望**从过去学习**并**预测未来**。也就是说，在划分训练集和测试集的时候，我们希望使用某个特定日期之前的所有数据作为训练集，该日期之后的所有数据作为测试集。这是我们通常使用时间序列预测的方式：已知过去所有的出租数据，我们认为明天会发生什么？我们将使用前 184 个数据点（对应前 23 天）作为训练集，剩余的 64 个数据点（对应剩余的 8 天）作为测试集。

在我们的预测任务中，我们使用的唯一特征就是某一租车数量对应的日期和时间。因此输入特征是日期和时间，比如 `2015-08-01 00:00:00`，而输出是在接下来 3 小时内的租车数量（根据我们的 `DataFrame`，在这个例子中是 3）。

在计算机上存储日期的常用方式是使用 POSIX 时间（这有些令人意外），它是从 1970 年 1 月 1 日 00:00:00（也就是 Unix 时间的起点）起至现在的总秒数。首先，我们可以尝试使用这个单一整数特征作为数据表示：

In[52]:

```
# 提取目标值（租车数量）
y = citibike.values
# 利用"%s"将时间转换为POSIX时间
X = citibike.index.strftime("%s").astype("int").reshape(-1, 1)
```

我们首先定义一个函数，它可以将数据划分为训练集和测试集，构建模型并将结果可视化：

In[54]:

```
# 使用前184个数据点用于训练，剩余的数据点用于测试
n_train = 184

# 对给定特征集上的回归进行评估和作图的函数
def eval_on_features(features, target, regressor):
    # 将给定特征划分为训练集和测试集
    X_train, X_test = features[:n_train], features[n_train:]
    # 同样划分目标数组
```

```
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)))
    y_pred = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)
    plt.figure(figsize=(10, 3))

    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90,
               ha="left")

    plt.plot(range(n_train), y_train, label="train")
    plt.plot(range(n_train, len(y_test) + n_train), y_test, '-', label="test")
    plt.plot(range(n_train), y_pred_train, '--', label="prediction train")

    plt.plot(range(n_train, len(y_test) + n_train), y_pred, '--',
             label="prediction test")
    plt.legend(loc=(1.01, 0))
    plt.xlabel("Date")
    plt.ylabel("Rentals")
```

我们之前看到，随机森林需要很少的数据预处理，因此它似乎很适合作为第一个模型。我们使用 POSIX 时间特征 X，并将随机森林回归传入我们的 eval_on_features 函数。结果如图 4-13 所示。

In[55]:

```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators=100, random_state=0)
plt.figure()
eval_on_features(X, y, regressor)
```

Out[55]:

```
Test-set R^2: -0.04
```

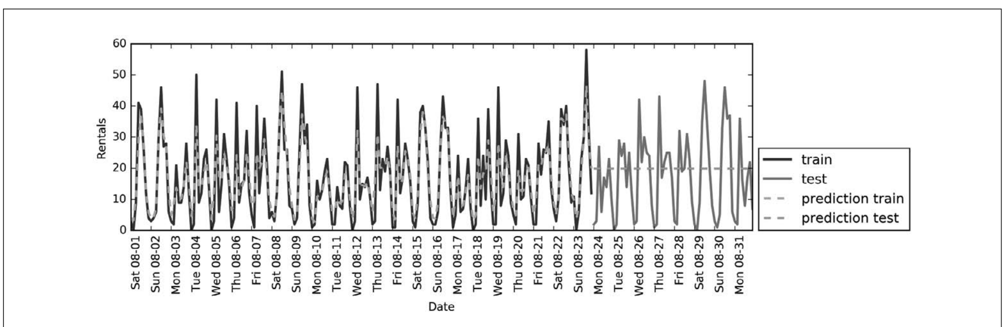


图 4-13：随机森林仅使用 POSIX 时间做出的预测

在训练集上的预测结果相当好，这符合随机森林通常的表现。但对于测试集来说，预测结果是一条常数直线。R^2 为 −0.04，说明我们什么都没有学到。发生了什么？

问题在于特征和随机森林的组合。测试集中 POSIX 时间特征的值超出了训练集中特征取值的范围：测试集中数据点的时间戳要晚于训练集中的所有数据点。树以及随机森林无法**外**

推（extrapolate）到训练集之外的特征范围。结果就是模型只能预测训练集中最近数据点的目标值，即最后一次观测到数据的时间。

显然，我们可以做得更好。这就是我们的“专家知识”的用武之地。通过观察训练数据中的租车数量图像，我们发现两个因素似乎非常重要：一天内的时间与一周的星期几。因此我们来添加这两个特征。我们从 POSIX 时间中学不到任何东西，所以删掉这个特征。首先，我们仅使用每天的时刻。如图 4-14 所示，现在的预测结果对一周内的每天都具有相同的模式：

In[56]:

```
X_hour = citibike.index.hour.reshape(-1, 1)
eval_on_features(X_hour, y, regressor)
```

Out[56]:

```
Test-set R^2: 0.60
```

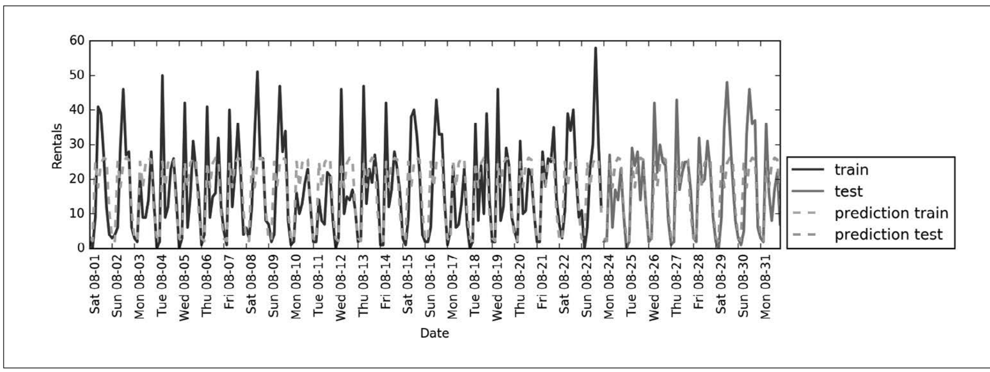


图 4-14：随机森林仅使用每天的时刻做出的预测

R^2 已经好多了，但预测结果显然没有抓住每周的模式。下面我们还添加一周的星期几作为特征（见图 4-15）：

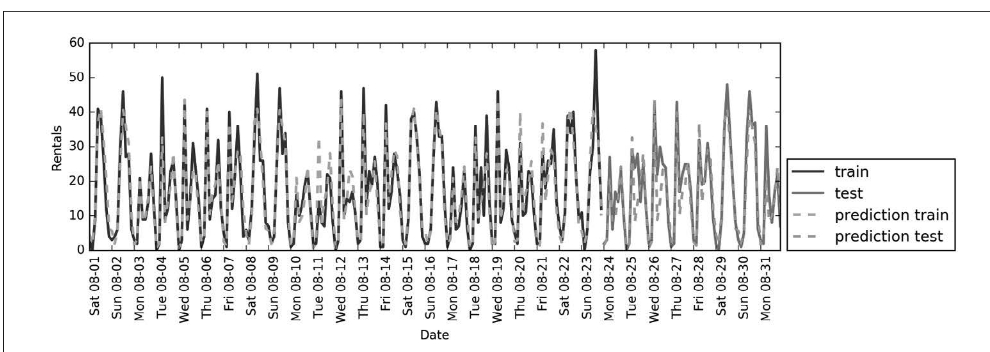


图 4-15：随机森林使用一周的星期几和每天的时刻两个特征做出的预测

In[57]:

```
X_hour_week = np.hstack([citibike.index.dayofweek.reshape(-1, 1),
                         citibike.index.hour.reshape(-1, 1)])
eval_on_features(X_hour_week, y, regressor)
```

Out[57]:

```
Test-set R^2: 0.84
```

现在我们的模型通过考虑一周的星期几和一天内的时间捕捉到了周期性的行为。它的 R^2 为 0.84，预测性能相当好。模型学到的内容可能是 8 月前 23 天中星期几与时刻每种组合的平均租车数量。这实际上不需要像随机森林这样复杂的模型，所以我们尝试一个更简单的模型——`LinearRegression`（见图 4-16）：

In[58]:

```
from sklearn.linear_model import LinearRegression
eval_on_features(X_hour_week, y, LinearRegression())
```

Out[58]:

```
Test-set R^2: 0.13
```

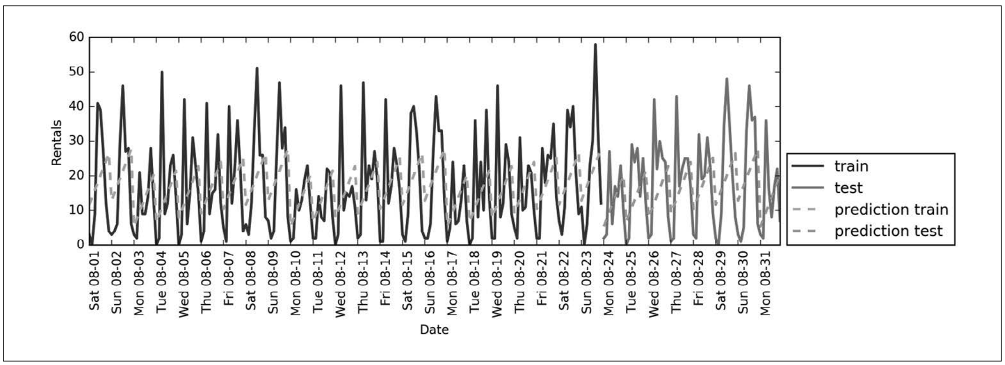


图 4-16：线性模型使用一周的星期几和每天的时刻两个特征做出的预测

`LinearRegression` 的效果差得多，而且周期性模式看起来很奇怪。其原因在于我们用整数编码一周的星期几和一天内的时间，它们被解释为连续变量。因此，线性模型只能学到关于每天时间的线性函数——它学到的是，时间越晚，租车数量越多。但实际模式比这要复杂得多。我们可以通过将整数解释为分类变量（用 `OneHotEncoder` 进行变换）来获取这种模式（见图 4-17）：

In[59]:

```
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
```

In[60]:

```
eval_on_features(X_hour_week_onehot, y, Ridge())
```

Out[60]:

```
Test-set R^2: 0.62
```

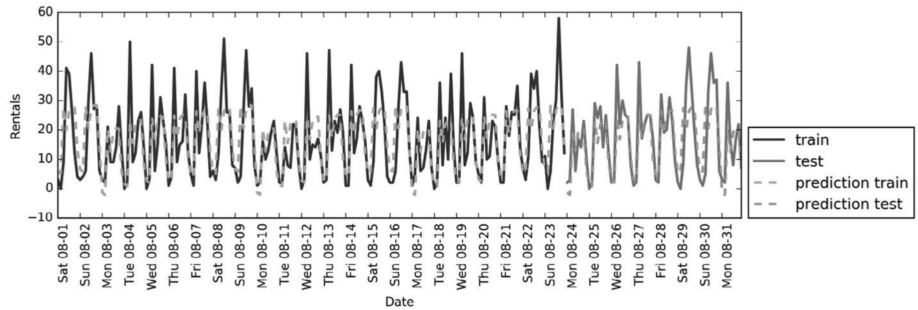


图 4-17：线性模型使用 one-hot 编码过的一周的星期几和每天的时刻两个特征做出的预测

它给出了比连续特征编码好得多的匹配。现在线性模型为一周内的每天都学到了一个系数，为一天内的每个时刻都学到了一个系数。也就是说，一周七天共享“一天内每个时刻”的模式。

利用交互特征，我们可以让模型为星期几和时刻的每一种组合学到一个系数（见图 4-18）：

In[61]:

```
poly_transformer = PolynomialFeatures(degree=2, interaction_only=True,
                                      include_bias=False)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

Out[61]:

```
Test-set R^2: 0.85
```

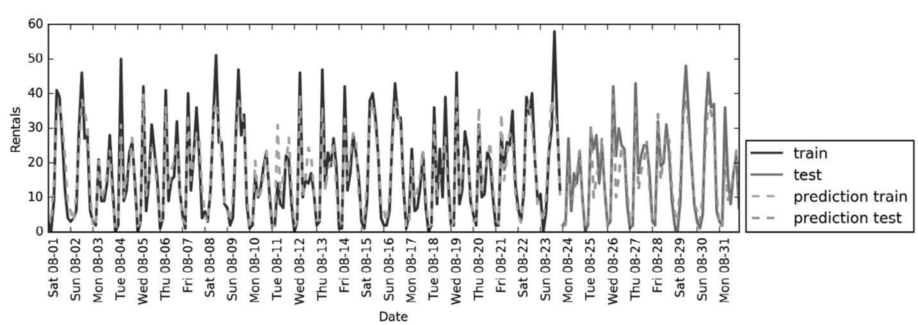


图 4-18：线性模型使用星期几和时刻两个特征的乘积做出的预测

这一变换最终得到一个性能与随机森林类似的模型。这个模型的一大优点是，可以很清楚地看到学到的内容：对每个星期几和时刻的交互项学到了一个系数。我们可以将模型学到的系数作图，而这对于随机森林来说是不可能的。

首先，为时刻和星期几特征创建特征名称：

In[62]:

```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features =  day + hour
```

然后，利用 get_feature_names 方法对 PolynomialFeatures 提取的所有交互特征进行命名，并仅保留系数不为零的那些特征：

In[63]:

```
features_poly = poly_transformer.get_feature_names(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

下面将线性模型学到的系数可视化，如图 4-19 所示：

In[64]:

```
plt.figure(figsize=(15, 2))
plt.plot(coef_nonzero, 'o')
plt.xticks(np.arange(len(coef_nonzero)), features_nonzero, rotation=90)
plt.xlabel("Feature name")
plt.ylabel("Feature magnitude")
```

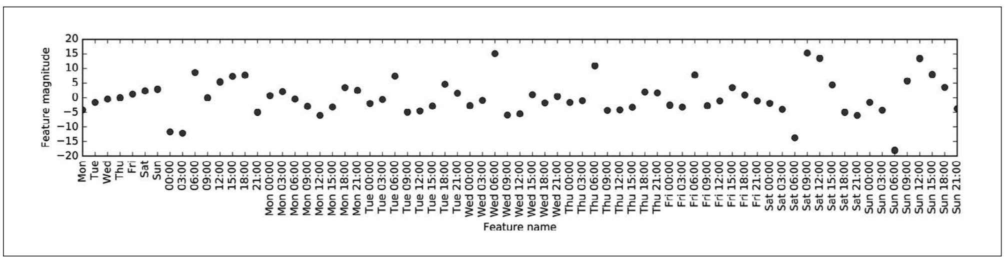


图 4-19：线性模型使用星期几和时刻两个特征的乘积学到的系数

4.7　小结与展望

本章讨论了如何处理不同的数据类型（特别是分类变量）。我们强调了使用适合机器学习算法的数据表示方式的重要性，例如 one-hot 编码过的分类变量。还讨论了通过特征工程生成新特征的重要性，以及利用专家知识从数据中创建导出特征的可能性。特别是线性模型，可能会从分箱、添加多项式和交互项而生成的新特征中大大受益。对于更加复杂的非线性模型（比如随机森林和 SVM），在无需显式扩展特征空间的前提下就可以学习更加复杂的任务。在实践中，所使用的特征（以及特征与方法之间的匹配）通常是使机器学习方法表现良好的最重要的因素。

现在你已经知道了如何适当地表示数据，以及对哪个任务使用哪种算法，下一章节重点介绍评估机器学习模型的性能与选择正确的参数设置。

第 5 章

模型评估与改进

前面讨论了监督学习和无监督学习的基本原理，并探索了多种机器学习算法，本章我们深入学习模型评估与参数选择。

我们将重点介绍监督方法，包括回归与分类，因为在无监督学习中，模型评估与选择通常是一个非常定性的过程（正如我们在第 3 章中所见）。

到目前为止，为了评估我们的监督模型，我们使用 `train_test_split` 函数将数据集划分为训练集和测试集，在训练集上调用 `fit` 方法来构建模型，并且在测试集上用 `score` 方法来评估这个模型——对于分类问题而言，就是计算正确分类的样本所占的比例。下面是这个过程的一个示例：

In[2]:

```
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# 创建一个模拟数据集
X, y = make_blobs(random_state=0)
# 将数据和标签划分为训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# 将模型实例化，并用它来拟合训练集
logreg = LogisticRegression().fit(X_train, y_train)
# 在测试集上评估该模型
print("Test set score: {:.2f}".format(logreg.score(X_test, y_test)))
```

Out[2]:

```
Test set score: 0.88
```

请记住，之所以将数据划分为训练集和测试集，是因为我们想要度量模型对前所未见的新

数据的**泛化**性能。我们对模型在训练集上的拟合效果不感兴趣，而是想知道模型对于训练过程中没有见过的数据的预测能力。

本章我们将从两个方面进行模型评估。我们首先介绍交叉验证，然后讨论评估分类和回归性能的方法，其中前者是一种更可靠的评估泛化性能的方法，后者是在默认度量（`score` 方法给出的精度和 R^2）之外的方法。

我们还将讨论网格搜索，这是一种调节监督模型参数以获得最佳泛化性能的有效方法。

5.1 交叉验证

交叉验证（cross-validation）是一种评估泛化性能的统计学方法，它比单次划分训练集和测试集的方法更加稳定、全面。在交叉验证中，数据被多次划分，并且需要训练多个模型。最常用的交叉验证是 ***k* 折交叉验证**（*k*-fold cross-validation），其中 *k* 是由用户指定的数字，通常取 5 或 10。在执行 5 折交叉验证时，首先将数据划分为（大致）相等的 5 部分，每一部分叫作**折**（fold）。接下来训练一系列模型。使用第 1 折作为测试集、其他折（2~5）作为训练集来训练第一个模型。利用 2~5 折中的数据来构建模型，然后在 1 折上评估精度。之后构建另一个模型，这次使用 2 折作为测试集，1、3、4、5 折中的数据作为训练集。利用 3、4、5 折作为测试集继续重复这一过程。对于将数据划分为训练集和测试集的这 5 次**划分**，每一次都要计算精度。最后我们得到了 5 个精度值。整个过程如图 5-1 所示。

In[3]:

```
mglearn.plots.plot_cross_validation()
```

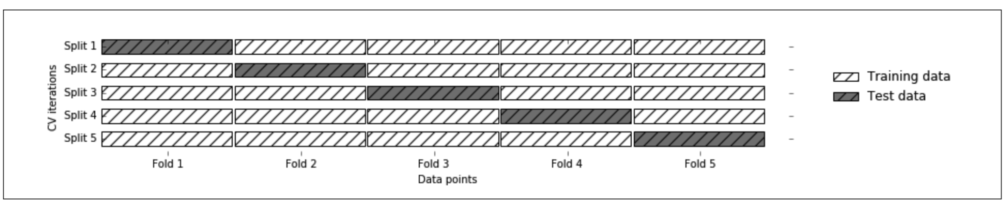


图 5-1：5 折交叉验证中的数据划分

通常来说，数据的前五分之一是第 1 折，第二个五分之一是第 2 折，以此类推。

5.1.1 scikit-learn中的交叉验证

`scikit-learn` 是利用 `model_selection` 模块中的 `cross_val_score` 函数来实现交叉验证的。`cross_val_score` 函数的参数是我们想要评估的模型、训练数据与真实标签。我们在 `iris` 数据集上对 `LogisticRegression` 进行评估：

In[4]:

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
```

```
logreg = LogisticRegression()

scores = cross_val_score(logreg, iris.data, iris.target)
print("Cross-validation scores: {}".format(scores))
```

Out[4]:

```
Cross-validation scores: [ 0.961  0.922  0.958]
```

默认情况下，`cross_val_score` 执行 3 折交叉验证，返回 3 个精度值。可以通过修改 `cv` 参数来改变折数：

In[5]:

```
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
print("Cross-validation scores: {}".format(scores))
```

Out[5]:

```
Cross-validation scores: [ 1.     0.967  0.933  0.9    1.   ]
```

总结交叉验证精度的一种常用方法是计算平均值：

In[6]:

```
print("Average cross-validation score: {:.2f}".format(scores.mean()))
```

Out[6]:

```
Average cross-validation score: 0.96
```

我们可以从交叉验证平均值中得出结论，我们预计模型的平均精度约为 96%。观察 5 折交叉验证得到的所有 5 个精度值，我们还可以发现，折与折之间的精度有较大的变化，范围为从 100% 精度到 90% 精度。这可能意味着模型强烈依赖于将某个折用于训练，但也可能只是因为数据集的数据量太小。

5.1.2　交叉验证的优点

使用交叉验证而不是将数据单次划分为训练集和测试集，这种做法具有下列优点。首先，`train_test_split` 对数据进行随机划分。想象一下，在随机划分数据时我们很“幸运”，所有难以分类的样例都在训练集中。在这种情况下，测试集将仅包含“容易分类的”样例，并且测试集精度会高得不切实际。相反，如果我们“不够幸运”，则可能随机地将所有难以分类的样例都放在测试集中，因此得到一个不切实际的低分数。但如果使用交叉验证，每个样例都会刚好在测试集中出现一次：每个样例位于一个折中，而每个折都在测试集中出现一次。因此，模型需要对数据集中所有样本的泛化能力都很好，才能让所有的交叉验证得分（及其平均值）都很高。

对数据进行多次划分，还可以提供我们的模型对训练集选择的敏感性信息。对于 `iris` 数据集，我们观察到精度在 90% 到 100% 之间。这是一个不小的范围，它告诉我们将模型应用于新数据时在最坏情况和最好情况下的可能表现。

与数据的单次划分相比，交叉验证的另一个优点是我们对数据的使用更加高效。在使用 `train_test_split` 时，我们通常将 75% 的数据用于训练，25% 的数据用于评估。在使用 5

折交叉验证时，在每次迭代中我们可以使用 4/5（80%）的数据来拟合模型。在使用 10 折交叉验证时，我们可以使用 9/10（90%）的数据来拟合模型。更多的数据通常可以得到更为精确的模型。

交叉验证的主要缺点是增加了计算成本。现在我们要训练 *k* 个模型而不是单个模型，所以交叉验证的速度要比数据的单次划分大约慢 *k* 倍。

> 重要的是要记住，交叉验证不是一种构建可应用于新数据的模型的方法。交叉验证不会返回一个模型。在调用 `cross_val_score` 时，内部会构建多个模型，但交叉验证的目的只是评估给定算法在特定数据集上训练后的泛化性能好坏。

5.1.3　分层*k*折交叉验证和其他策略

将数据集划分为 *k* 折时，从数据的前 *k* 分之一开始划分（正如上一节所述），这可能并不总是一个好主意。例如，我们来看一下 `iris` 数据集：

In[7]:

```
from sklearn.datasets import load_iris
iris = load_iris()
print("Iris labels:\n{}".format(iris.target))
```

Out[7]:

```
Iris labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

如你所见，数据的前三分之一是类别 0，中间三分之一是类别 1，最后三分之一是类别 2。想象一下在这个数据集上进行 3 折交叉验证。第 1 折将只包含类别 0，所以在数据的第一次划分中，测试集将只包含类别 0，而训练集只包含类别 1 和 2。由于在 3 次划分中训练集和测试集中的类别都不相同，因此这个数据集上的 3 折交叉验证精度为 0。这没什么帮助，因为我们在 `iris` 上可以得到比 0% 好得多的精度。

由于简单的 *k* 折策略在这里失效了，所以 `scikit-learn` 在分类问题中不使用这种策略，而是使用**分层 *k* 折交叉验证**（stratified *k*-fold cross-validation）。在分层交叉验证中，我们划分数据，使每个折中类别之间的比例与整个数据集中的比例相同，如图 5-2 所示。

In[8]:

```
mglearn.plots.plot_stratified_cross_validation()
```

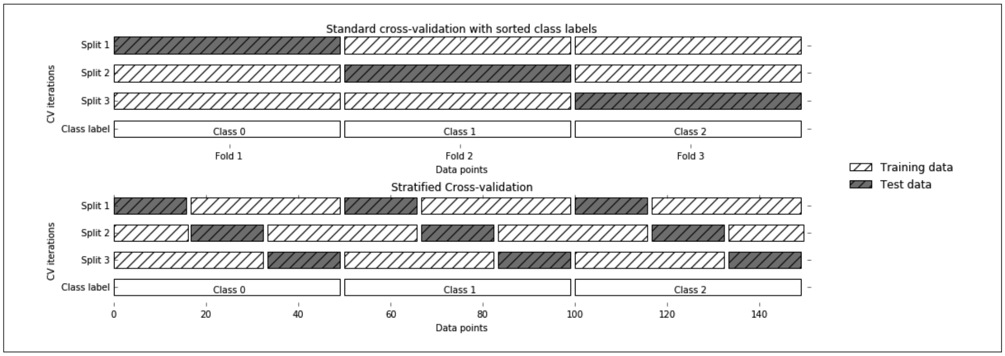


图 5-2：当数据按类别标签排序时，标准交叉验证与分层交叉验证的对比

举个例子，如果 90% 的样本属于类别 A 而 10% 的样本属于类别 B，那么分层交叉验证可以确保，在每个折中 90% 的样本属于类别 A 而 10% 的样本属于类别 B。

使用分层 k 折交叉验证而不是 k 折交叉验证来评估一个分类器，这通常是一个好主意，因为它可以对泛化性能做出更可靠的估计。在只有 10% 的样本属于类别 B 的情况下，如果使用标准 k 折交叉验证，很可能某个折中只包含类别 A 的样本。利用这个折作为测试集的话，无法给出分类器整体性能的信息。

对于回归问题，`scikit-learn` 默认使用标准 k 折交叉验证。也可以尝试让每个折表示回归目标的不同取值，但这并不是一种常用的策略，也会让大多数用户感到意外。

1. 对交叉验证的更多控制

我们之前看到，可以利用 `cv` 参数来调节 `cross_val_score` 所使用的折数。但 `scikit-learn` 允许提供一个**交叉验证分离器**（cross-validation splitter）作为 `cv` 参数，来对数据划分过程进行更精细的控制。对于大多数使用场景而言，回归问题默认的 k 折交叉验证与分类问题的分层 k 折交叉验证的表现都很好，但有些情况下你可能希望使用不同的策略。比如说，我们想要在一个分类数据集上使用标准 k 折交叉验证来重现别人的结果。为了实现这一点，我们首先必须从 `model_selection` 模块中导入 `KFold` 分离器类，并用我们想要使用的折数来将其实例化：

In[9]:

```
from sklearn.model_selection import KFold
kfold = KFold(n_splits=5)
```

然后我们可以将 `kfold` 分离器对象作为 `cv` 参数传入 `cross_val_score`：

In[10]:

```
print("Cross-validation scores:\n{}".format(
      cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[10]:

```
Cross-validation scores:
[ 1.     0.933  0.433  0.967  0.433]
```

通过这种方法，我们可以验证，在 `iris` 数据集上使用 3 折交叉验证（不分层）确实是一个非常糟糕的主意：

In[11]:

```
kfold = KFold(n_splits=3)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[11]:

```
Cross-validation scores:
[ 0.  0.  0.]
```

请记住，在 `iris` 数据集中每个折对应一个类别，因此学不到任何内容。解决这个问题的另一种方法是将数据打乱来代替分层，以打乱样本按标签的排序。可以通过将 `KFold` 的 `shuffle` 参数设为 `True` 来实现这一点。如果我们将数据打乱，那么还需要固定 `random_state` 以获得可重复的打乱结果。否则，每次运行 `cross_val_score` 将会得到不同的结果，因为每次使用的是不同的划分（这可能并不是一个问题，但可能会出人意料）。在划分数据之前将其打乱可以得到更好的结果：

In[12]:

```
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[12]:

```
Cross-validation scores:
[ 0.9   0.96  0.96]
```

2. 留一法交叉验证

另一种常用的交叉验证方法是**留一法**（leave-one-out）。你可以将留一法交叉验证看作是每折只包含单个样本的 k 折交叉验证。对于每次划分，你选择单个数据点作为测试集。这种方法可能非常耗时，特别是对于大型数据集来说，但在小型数据集上有时可以给出更好的估计结果：

In[13]:

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Number of cv iterations: ", len(scores))
print("Mean accuracy: {:.2f}".format(scores.mean()))
```

Out[13]:

```
Number of cv iterations:  150
Mean accuracy: 0.95
```

3. 打乱划分交叉验证

另一种非常灵活的交叉验证策略是**打乱划分交叉验证**（shuffle-split cross-validation）。在打乱划分交叉验证中，每次划分为训练集取样 `train_size` 个点，为测试集取样 `test_size` 个（不相交的）点。将这一划分方法重复 `n_iter` 次。图 5-3 显示的是对包含 10 个点的数据集

运行 4 次迭代划分，每次的训练集包含 5 个点，测试集包含 2 个点（你可以将 train_size 和 test_size 设为整数来表示这两个集合的绝对大小，也可以设为浮点数来表示占整个数据集的比例）：

In[14]:

```
mglearn.plots.plot_shuffle_split()
```

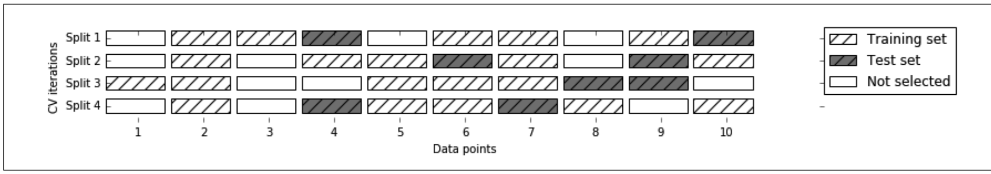


图 5-3：对 10 个点进行打乱划分，其中 train_size=5、test_size=2、n_iter=4

下面的代码将数据集划分为 50% 的训练集和 50% 的测试集，共运行 10 次迭代[1]：

In[15]:

```
from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Cross-validation scores:\n{}".format(scores))
```

Out[15]:

```
Cross-validation scores:
[ 0.96   0.907  0.947  0.96   0.96   0.907  0.893  0.907  0.92   0.973]
```

打乱划分交叉验证可以在训练集和测试集大小之外独立控制迭代次数，这有时是很有帮助的。它还允许在每次迭代中仅使用部分数据，这可以通过设置 train_size 与 test_size 之和不等于 1 来实现。用这种方法对数据进行二次采样可能对大型数据上的试验很有用。

ShuffleSplit 还有一种分层的形式，其名称为 StratifiedShuffleSplit，它可以为分类任务提供更可靠的结果。

4. 分组交叉验证

另一种非常常见的交叉验证适用于数据中的分组高度相关时。比如你想构建一个从人脸图片中识别情感的系统，并且收集了 100 个人的照片的数据集，其中每个人都进行了多次拍摄，分别展示了不同的情感。我们的目标是构建一个分类器，能够正确识别未包含在数据集中的人的情感。你可以使用默认的分层交叉验证来度量分类器的性能。但是这样的话，同一个人的照片可能会同时出现在训练集和测试集中。对于分类器而言，检测训练集中出现过的人脸情感比全新的人脸要容易得多。因此，为了准确评估模型对新的人脸的泛化能力，我们必须确保训练集和测试集中包含不同人的图像。

为了实现这一点，我们可以使用 GroupKFold，它以 groups 数组作为参数，可以用来说明照片中对应的是哪个人。这里的 groups 数组表示数据中的分组，在创建训练集和测试集的时候不应该将其分开，也不应该与类别标签弄混。

注 1：由于此处 ShuffleSplit 未设置 random_state，因此实际运行结果可能和书中有所不同。——译者注

数据分组的这种例子常见于医疗应用，你可能拥有来自同一名病人的多个样本，但想要将其泛化到新的病人。同样，在语音识别领域，你的数据集中可能包含同一名发言人的多条记录，但你希望能够识别新的发言人的讲话。

下面这个示例用到了一个由 `groups` 数组指定分组的模拟数据集。这个数据集包含 12 个数据点，且对于每个数据点，`groups` 指定了该点所属的分组（想想病人的例子）。一共分成 4 个组，前 3 个样本属于第一组，接下来的 4 个样本属于第二组，以此类推：

In[16]:

```
from sklearn.model_selection import GroupKFold
# 创建模拟数据集
X, y = make_blobs(n_samples=12, random_state=0)
# 假设前3个样本属于同一组，接下来的4个属于同一组，以此类推
groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Cross-validation scores:\n{}".format(scores))
```

Out[16]:

```
Cross-validation scores:
[ 0.75   0.8    0.667]
```

样本不需要按分组进行排序，我们这么做只是为了便于说明。基于这些标签计算得到的划分如图 5-4 所示。

如你所见，对于每次划分，每个分组都是整体出现在训练集或测试集中：

In[17]:

```
mglearn.plots.plot_group_kfold()
```

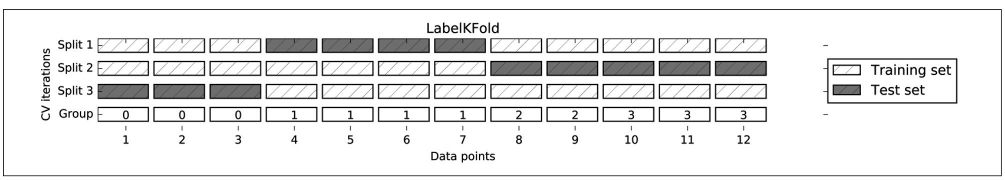


图 5-4：用 GroupKFold 进行依赖于标签的划分

`scikit-learn` 中还有很多交叉验证的划分策略，适用于更多的使用场景 [你可以在 `scikit-learn` 的用户指南页面查看这些内容（http://scikit-learn.org/stable/modules/cross_validation.html）]。但标准的 `KFold`、`StratifiedKFold` 和 `GroupKFold` 是目前最常用的几种。

5.2 网格搜索

现在我们知道了如果评估一个模型的泛化能力，下面继续学习通过调参来提升模型的泛化性能。第 2 章和第 3 章中讨论过 `scikit-learn` 中许多算法的参数设置，在尝试调参之前，重要的是要理解参数的含义。找到一个模型的重要参数（提供最佳泛化性能的参数）的取值是一项棘手的任务，但对于几乎所有模型和数据集来说都是必要的。由于这项任务如此常见，所以 `scikit-learn` 中有一些标准方法可以帮你完成。最常用的方法就是**网格搜索**

（grid search），它主要是指尝试我们关心的参数的所有可能组合。

考虑一个具有 RBF（径向基函数）核的核 SVM 的例子，它在 `SVC` 类中实现。正如第 2 章中所述，它有 2 个重要参数：核宽度 `gamma` 和正则化参数 `C`。假设我们希望尝试 `C` 的取值为 `0.001`、`0.01`、`0.1`、`1`、`10` 和 `100`，`gamma` 也取这 6 个值。由于我想要尝试的 `C` 和 `gamma` 都有 6 个不同的取值，所以总共有 36 种参数组合。所有可能的组合组成了 SVM 的参数设置表（网格），如下所示。

	C=0.001	C=0.01	...	C=10
gamma=0.001	SVC(C=0.001, gamma=0.001)	SVC(C=0.01, gamma=0.001)	...	SVC(C=10, gamma=0.001)
gamma=0.01	SVC(C=0.001, gamma=0.01)	SVC(C=0.01, gamma=0.01)	...	SVC(C=10, gamma=0.01)
...	...	...	...	...
gamma=100	SVC(C=0.001, gamma=100)	SVC(C=0.01, gamma=100)	...	SVC(C=10, gamma=100)

5.2.1 简单网格搜索

我们可以实现一个简单的网格搜索，在 2 个参数上使用 `for` 循环，对每种参数组合分别训练并评估一个分类器：

In[18]:

```
# 简单的网格搜索实现
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
print("Size of training set: {}   size of test set: {}".format(
      X_train.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # 对每种参数组合都训练一个SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # 在测试集上评估SVC
        score = svm.score(X_test, y_test)
        # 如果我们得到了更高的分数，则保存该分数和对应的参数
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("Best score: {:.2f}".format(best_score))
print("Best parameters: {}".format(best_parameters))
```

Out[18]:

```
Size of training set: 112   size of test set: 38
Best score: 0.97
Best parameters: {'C': 100, 'gamma': 0.001}
```

5.2.2 参数过拟合的风险与验证集

看到这个结果，我们可能忍不住要报告，我们找到了一个在数据集上精度达到 97% 的模型。然而，这种说法可能过于乐观了（或者就是错的），其原因如下：我们尝试了许多不同的参数，并选择了在测试集上精度最高的那个，但这个精度不一定能推广到新数据上。由于我们使用测试数据进行调参，所以不能再用它来评估模型的好坏。我们最开始需要将数据划分为训练集和测试集也是因为这个原因。我们需要一个独立的数据集来进行评估，一个在创建模型时没有用到的数据集。

为了解决这个问题，一种方法是再次划分数据，这样我们得到 3 个数据集：用于构建模型的训练集，用于选择模型参数的验证集（开发集），用于评估所选参数性能的测试集。图 5-5 给出了这 3 个集合的图示：

In[19]:

```
mglearn.plots.plot_threefold_split()
```

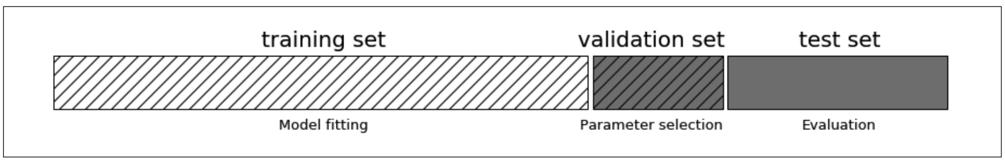


图 5-5：对数据进行 3 折划分，分为训练集、验证集和测试集

利用验证集选定最佳参数之后，我们可以利用找到的参数设置重新构建一个模型，但是要同时在训练数据和验证数据上进行训练。这样我们可以利用尽可能多的数据来构建模型。其实现如下所示：

In[20]:

```
from sklearn.svm import SVC
# 将数据划分为训练+验证集与测试集
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
# 将训练+验证集划分为训练集与验证集
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
print("Size of training set: {}   size of validation set: {}   size of test set:"
      " {}\n".format(X_train.shape[0], X_valid.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # 对每种参数组合都训练一个SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # 在验证集上评估SVC
        score = svm.score(X_valid, y_valid)
        # 如果我们得到了更高的分数，则保存该分数和对应的参数
        if score > best_score:
```

```
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
# 在训练+验证集上重新构建一个模型，并在测试集上进行评估
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
test_score = svm.score(X_test, y_test)
print("Best score on validation set: {:.2f}".format(best_score))
print("Best parameters: ", best_parameters)
print("Test set score with best parameters: {:.2f}".format(test_score))
```

Out[20]:

```
Size of training set: 84   size of validation set: 28   size of test set: 38

Best score on validation set: 0.96
Best parameters:  {'C': 10, 'gamma': 0.001}
Test set score with best parameters: 0.92
```

验证集上的最高分数是 96%，这比之前略低，可能是因为我们使用了更少的数据来训练模型（现在 `X_train` 更小，因为我们对数据集做了两次划分）。但测试集上的分数（这个分数实际反映了模型的泛化能力）更低，为 92%。因此，我们只能声称对 92% 的新数据正确分类，而不是我们之前认为的 97%！

训练集、验证集和测试集之间的区别对于在实践中应用机器学习方法至关重要。任何根据测试集精度所做的选择都会将测试集的信息“泄漏”（leak）到模型中。因此，保留一个单独的测试集是很重要的，它仅用于最终评估。好的做法是利用训练集和验证集的组合完成所有的探索性分析与模型选择，并保留测试集用于最终评估——即使对于探索性可视化也是如此。严格来说，在测试集上对不止一个模型进行评估并选择更好的那个，将会导致对模型精度过于乐观的估计。

5.2.3　带交叉验证的网格搜索

虽然将数据划分为训练集、验证集和测试集的方法（如上所述）是可行的，也相对常用，但这种方法对数据的划分方法相当敏感。从上面代码片段的输出中可以看出，网格搜索选择 `'C': 10, 'gamma': 0.001` 作为最佳参数，而 5.2.2 节的代码输出选择 `'C': 100, 'gamma': 0.001` 作为最佳参数。为了得到对泛化性能的更好估计，我们可以使用交叉验证来评估每种参数组合的性能，而不是仅将数据单次划分为训练集与验证集。这种方法用代码表示如下：

In[21]:

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # 对于每种参数组合都训练一个SVC
        svm = SVC(gamma=gamma, C=C)
        # 执行交叉验证
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # 计算交叉验证平均精度
        score = np.mean(scores)
        # 如果我们得到了更高的分数，则保存该分数和对应的参数
        if score > best_score:
```

```
                best_score = score
                best_parameters = {'C': C, 'gamma': gamma}
# 在训练+验证集上重新构建一个模型
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

要想使用 5 折交叉验证对 `C` 和 `gamma` 特定取值的 SVM 的精度进行评估，需要训练 $36 \times 5 = 180$ 个模型。你可以想象，使用交叉验证的主要缺点就是训练所有这些模型所需花费的时间。

下面的可视化（图 5-6）说明了上述代码如何选择最佳参数设置：

In[22]:

```
mglearn.plots.plot_cross_val_selection()
```

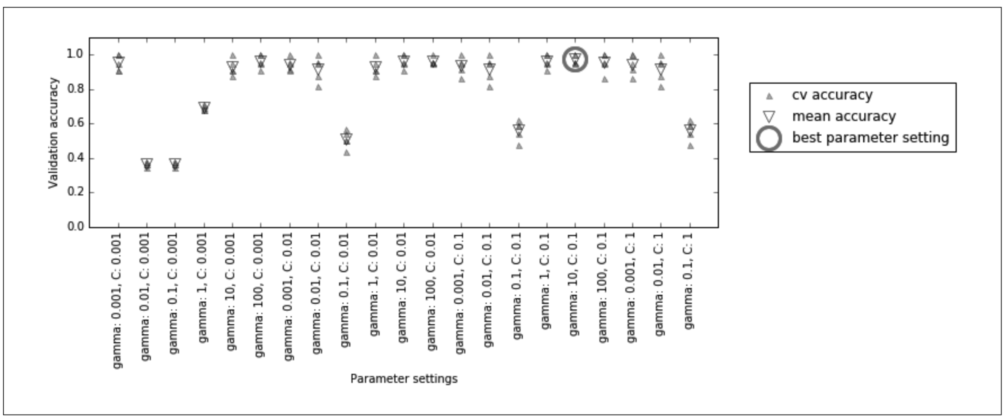


图 5-6：带交叉验证的网格搜索结果

对于每种参数设置（图中仅显示了一部分），需要计算 5 个精度值，交叉验证的每次划分都要计算一个精度值。然后，对每种参数设置计算平均验证精度。最后，选择平均验证精度最高的参数，用圆圈标记。

> 如前所述，交叉验证是在特定数据集上对给定算法进行评估的一种方法。但它通常与网格搜索等参数搜索方法结合使用。因此，许多人使用**交叉验证**（cross-validation）这一术语来通俗地指代带交叉验证的网格搜索。

划分数据、运行网格搜索并评估最终参数，这整个过程如图 5-7 所示。

In[23]:

```
mglearn.plots.plot_grid_search_overview()
```

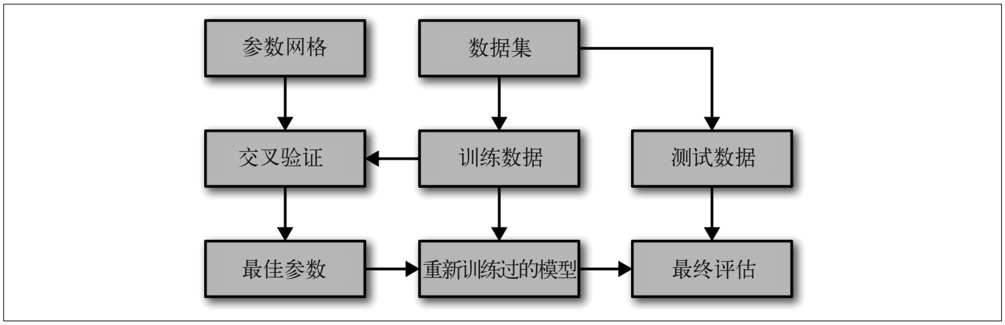


图 5-7：用 GridSearchCV 进行参数选择与模型评估的过程概述

由于带交叉验证的网格搜索是一种常用的调参方法，因此 scikit-learn 提供了 GridSearchCV 类，它以估计器（estimator）的形式实现了这种方法。要使用 GridSearchCV 类，你首先需要用一个字典指定要搜索的参数。然后 GridSearchCV 会执行所有必要的模型拟合。字典的键是我们要调节的参数名称（在构建模型时给出，在这个例子中是 C 和 gamma），字典的值是我们想要尝试的参数设置。如果 C 和 gamma 想要尝试的取值为 0.001、0.01、0.1、1、10 和 100，可以将其转化为下面这个字典：

In[24]:

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))
```

Out[24]:

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

现在我们可以使用模型（SVC）、要搜索的参数网格（param_grid）与要使用的交叉验证策略（比如 5 折分层交叉验证）将 GridSearchCV 类实例化：

In[25]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

GridSearchCV 将使用交叉验证来代替之前用过的划分训练集和验证集方法。但是，我们仍需要将数据划分为训练集和测试集，以避免参数过拟合：

In[26]:

```
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
```

我们创建的 grid_search 对象的行为就像是一个分类器，我们可以对它调用标准的 fit、predict 和 score 方法。[2] 但我们在调用 fit 时，它会对 param_grid 指定的每种参数组合都

注 2：用另一个估计器创建的 scikit-learn 估计器被称为**元估计器**（meta-estimator）。GridSearchCV 是最常用的元估计器，但后面我们将看到更多。

运行交叉验证：

In[27]:
```
grid_search.fit(X_train, y_train)
```

拟合 GridSearchCV 对象不仅会搜索最佳参数，还会利用得到最佳交叉验证性能的参数在整个训练数据集上自动拟合一个新模型。因此，fit 完成的工作相当于本节开头 In[21] 的代码结果。GridSearchCV 类提供了一个非常方便的接口，可以用 predict 和 score 方法来访问重新训练过的模型。为了评估找到的最佳参数的泛化能力，我们可以在测试集上调用 score：

In[28]:
```
print("Test set score: {:.2f}".format(grid_search.score(X_test, y_test)))
```

Out[28]:
```
Test set score: 0.97
```

利用交叉验证选择参数，我们实际上找到了一个在测试集上精度为 97% 的模型。重要的是，我们**没有使用测试集**来选择参数。我们找到的参数保存在 best_params_ 属性中，而交叉验证最佳精度（对于这种参数设置，不同划分的平均精度）保存在 best_score_ 中：

In[29]:
```
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

Out[29]:
```
Best parameters: {'C': 100, 'gamma': 0.01}
Best cross-validation score: 0.97
```

同样，注意不要将 best_score_ 与模型在测试集上调用 score 方法计算得到的泛化性能弄混。使用 score 方法（或者对 predict 方法的输出进行评估）采用的是**在整个训练集上训练**的模型。而 best_score_ 属性保存的是交叉验证的平均精度，是**在训练集上进行交叉验证**得到的。

能够访问实际找到的模型，这有时是很有帮助的，比如查看系数或特征重要性。你可以用 best_estimator_ 属性来访问最佳参数对应的模型，它是在整个训练集上训练得到的：

In[30]:
```
print("Best estimator:\n{}".format(grid_search.best_estimator_))
```

Out[30]:
```
Best estimator:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
   decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',
   max_iter=-1, probability=False, random_state=None, shrinking=True,
   tol=0.001, verbose=False)
```

由于 grid_search 本身具有 predict 和 score 方法，所以不需要使用 best_estimator_ 来进行预测或评估模型。

1. 分析交叉验证的结果

将交叉验证的结果可视化通常有助于理解模型泛化能力对所搜索参数的依赖关系。由于运行网格搜索的计算成本相当高，所以通常最好从相对比较稀疏且较小的网格开始搜索。然后我们可以检查交叉验证网格搜索的结果，可能也会扩展搜索范围。网格搜索的结果可以在 cv_results_ 属性中找到，它是一个字典，其中保存了搜索的所有内容。你可以在下面的输出中看到，它里面包含许多细节，最好将其转换成 pandas 数据框后再查看：

In[31]:

```
import pandas as pd
# 转换为DataFrame（数据框）
results = pd.DataFrame(grid_search.cv_results_)
# 显示前5行
display(results.head())
```

Out[31]:

	param_C	param_gamma	params	mean_test_score
0	0.001	0.001	{'C': 0.001, 'gamma': 0.001}	0.366
1	0.001	0.01	{'C': 0.001, 'gamma': 0.01}	0.366
2	0.001	0.1	{'C': 0.001, 'gamma': 0.1}	0.366
3	0.001	1	{'C': 0.001, 'gamma': 1}	0.366
4	0.001	10	{'C': 0.001, 'gamma': 10}	0.366

	rank_test_score	split0_test_score	split1_test_score	split2_test_score
0	22	0.375	0.347	0.363
1	22	0.375	0.347	0.363
2	22	0.375	0.347	0.363
3	22	0.375	0.347	0.363
4	22	0.375	0.347	0.363

	split3_test_score	split4_test_score	std_test_score
0	0.363	0.380	0.011
1	0.363	0.380	0.011
2	0.363	0.380	0.011
3	0.363	0.380	0.011
4	0.363	0.380	0.011

results 中每一行对应一种特定的参数设置。对于每种参数设置，交叉验证所有划分的结果都被记录下来，所有划分的平均值和标准差也被记录下来。由于我们搜索的是一个二维参数网格（C 和 gamma），所以最适合用热图可视化（见图 5-8）。我们首先提取平均验证分数，然后改变分数数组的形状，使其坐标轴分别对应于 C 和 gamma：

In[32]:

```
scores = np.array(results.mean_test_score).reshape(6, 6)

# 对交叉验证平均分数作图
mglearn.tools.heatmap(scores, xlabel='gamma', xticklabels=param_grid['gamma'],
                      ylabel='C', yticklabels=param_grid['C'], cmap="viridis")
```

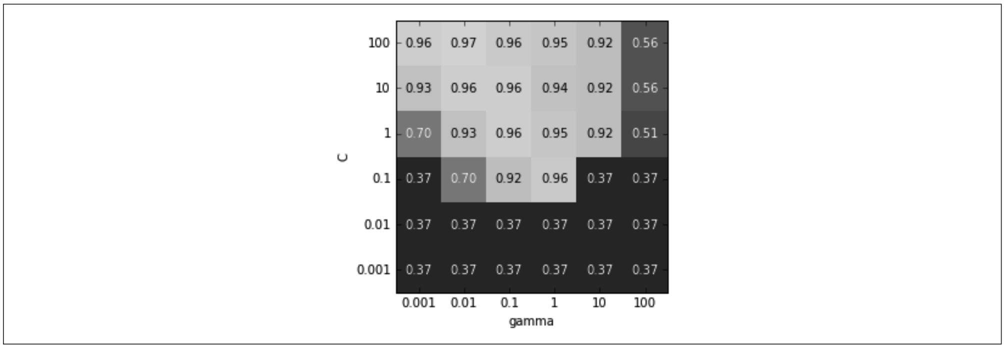


图 5-8：以 C 和 gamma 为自变量，交叉验证平均分数的热图

热图中的每个点对应于运行一次交叉验证以及一种特定的参数设置。颜色表示交叉验证的精度：浅色表示高精度，深色表示低精度。你可以看到，SVC 对参数设置非常敏感。对于许多种参数设置，精度都在 40% 左右，这是非常糟糕的；对于其他参数设置，精度约为 96%。我们可以从这张图中看出以下几点。首先，我们调节的参数对于获得良好的性能**非常重要**。这两个参数（C 和 gamma）都很重要，因为调节它们可以将精度从 40% 提高到 96%。此外，在我们选择的参数范围中也可以看到输出发生了显著的变化。同样重要的是要注意，参数的范围要足够大：每个参数的最佳取值不能位于图像的边界上。

下面我们来看几张图（见图 5-9），其结果不那么理想，因为选择的搜索范围不合适。

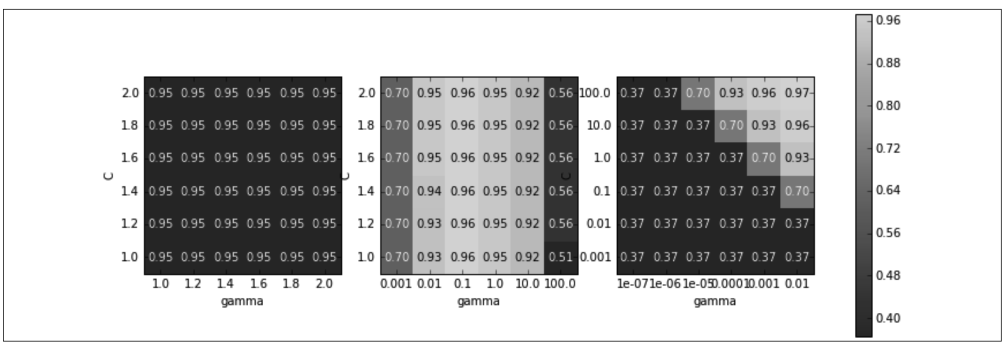


图 5-9：错误的搜索网格的热图可视化

In[33]:
```
fig, axes = plt.subplots(1, 3, figsize=(13, 5))

param_grid_linear = {'C': np.linspace(1, 2, 6),
                     'gamma':  np.linspace(1, 2, 6)}

param_grid_one_log = {'C': np.linspace(1, 2, 6),
                      'gamma':  np.logspace(-3, 2, 6)}

param_grid_range = {'C': np.logspace(-3, 2, 6),
                    'gamma':  np.logspace(-7, -2, 6)}
```

```
for param_grid, ax in zip([param_grid_linear, param_grid_one_log,
                           param_grid_range], axes):
    grid_search = GridSearchCV(SVC(), param_grid, cv=5)
    grid_search.fit(X_train, y_train)
    scores = grid_search.cv_results_['mean_test_score'].reshape(6, 6)

    # 对交叉验证平均分数作图
    scores_image = mglearn.tools.heatmap(
        scores, xlabel='gamma', ylabel='C', xticklabels=param_grid['gamma'],
        yticklabels=param_grid['C'], cmap="viridis", ax=ax)

plt.colorbar(scores_image, ax=axes.tolist())
```

第一张图没有显示任何变化，整个参数网格的颜色相同。在这种情况下，这是由参数 C 和 gamma 不正确的缩放以及不正确的范围造成的。但如果对于不同的参数设置都看不到精度的变化，也可能是因为这个参数根本不重要。通常最好在开始时尝试非常极端的值，以观察改变参数是否会导致精度发生变化。

第二张图显示的是垂直条形模式。这表示只有 gamma 的设置对精度有影响。这可能意味着 gamma 参数的搜索范围是我们所关心的，而 C 参数并不是——也可能意味着 C 参数并不重要。

第三张图中 C 和 gamma 对应的精度都有变化。但可以看到，在图像的整个左下角都没有发生什么有趣的事情。我们在后面的网格搜索中可以不考虑非常小的值。最佳参数设置出现在右上角。由于最佳参数位于图像的边界，所以我们可以认为，在这个边界之外可能还有更好的取值，我们可能希望改变搜索范围以包含这一区域内的更多参数。

基于交叉验证分数来调节参数网格是非常好的，也是探索不同参数的重要性的好方法。但是，你不应该在最终测试集上测试不同的参数范围——前面说过，只有确切知道了想要使用的模型，才能对测试集进行评估。

2. 在非网格的空间中搜索

在某些情况下，尝试所有参数的所有可能组合（正如 GridSearchCV 所做的那样）并不是一个好主意。例如，SVC 有一个 kernel 参数，根据所选择的 kernel（内核），其他参数也是与之相关的。如果 kernel='linear'，那么模型是线性的，只会用到 C 参数。如果 kernel='rbf'，则需要使用 C 和 gamma 两个参数（但用不到类似 degree 的其他参数）。在这种情况下，搜索 C、gamma 和 kernel 所有可能的组合没有意义：如果 kernel='linear'，那么 gamma 是用不到的，尝试 gamma 的不同取值将会浪费时间。为了处理这种“条件”（conditional）参数，GridSearchCV 的 param_grid 可以是字典组成的列表（a list of dictionaries）。列表中的每个字典可扩展为一个独立的网格。包含内核与参数的网格搜索可能如下所示。

In[34]:

```
param_grid = [{'kernel': ['rbf'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100],
               'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
              {'kernel': ['linear'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
print("List of grids:\n{}".format(param_grid))
```

Out[34]:

```
List of grids:
[{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100],
  'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
 {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

在第一个网格中，kernel 参数始终等于 'rbf'（注意 kernel 是一个长度为 1 的列表），而 C 和 gamma 都是变化的。在第二个网格中，kernel 参数始终等于 'linear'，只有 C 是变化的。下面我们来应用这个更加复杂的参数搜索：

In[35]:

```
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

Out[35]:

```
Best parameters: {'C': 100, 'kernel': 'rbf', 'gamma': 0.01}
Best cross-validation score: 0.97
```

我们再次查看 cv_results_。正如所料，如果 kernel 等于 'linear'，那么只有 C 是变化的[3]：

In[36]:

```
results = pd.DataFrame(grid_search.cv_results_)
# 我们给出的是转置后的表格，这样更适合页面显示：
display(results.T)
```

Out[36]:

	0	1	2	3	...	38	39	40	41
param_C	0.001	0.001	0.001	0.001	...	0.1	1	10	100
param_gamma	0.001	0.01	0.1	1	...	NaN	NaN	NaN	NaN
param_kernel	rbf	rbf	rbf	rbf	...	linear	linear	linear	linear
params	{*C*: 0.001, *kernel*: *rbf*, *gamma*: 0.001}	{*C*: 0.001, *kernel*: *rbf*, *gamma*: 0.01}	{*C*: 0.001, *kernel*: *rbf*, *gamma*: 0.1}	{*C*: 0.001, *kernel*: *rbf*, *gamma*: 1}	...	{*C*: 0.1, *kernel*: *linear*}	{*C*: 1, *kernel*: *linear*}	{*C*: 10, *kernel*: *linear*}	{*C*: 100, *kernel*: *linear*}
mean_test_score	0.37	0.37	0.37	0.37	...	0.95	0.97	0.96	0.96
rank_test_score	27	27	27	27	...	11	1	3	3
split0_test_score	0.38	0.38	0.38	0.38	...	0.96	1	0.96	0.96
split1_test_score	0.35	0.35	0.35	0.35	...	0.91	0.96	1	1
split2_test_score	0.36	0.36	0.36	0.36	...	1	1	1	1
split3_test_score	0.36	0.36	0.36	0.36	...	0.91	0.95	0.91	0.91
split4_test_score	0.38	0.38	0.38	0.38	...	0.95	0.95	0.95	0.95
std_test_score	0.011	0.011	0.011	0.011	...	0.033	0.022	0.034	0.034

```
12 rows × 42 columns
```

注 3：这里的实际输出结果有 23 行，作者在这里对其做了删节。详见本书 GitHub 仓库 https://github.com/amueller/introduction_to_ml_with_python/blob/master/05-model-evaluation-and-improvement.ipynb 的 In [36]。——译者注

3. 使用不同的交叉验证策略进行网格搜索

与 cross_val_score 类似，GridSearchCV 对分类问题默认使用分层 k 折交叉验证，对回归问题默认使用 k 折交叉验证。但是，你可以传入任何交叉验证分离器作为 GridSearchCV 的 cv 参数，正如 2.1.3 节中的“对交叉验证的更多控制”部分所述。特别地，如果只想将数据单次划分为训练集和验证集，你可以使用 ShuffleSplit 或 StratifiedShuffleSplit，并设置 n_iter=1。这对于非常大的数据集或非常慢的模型可能会有帮助。

(1) 嵌套交叉验证

在前面的例子中，我们先介绍了将数据单次划分为训练集、验证集与测试集，然后介绍了先将数据划分为训练集和测试集，再在训练集上进行交叉验证。但前面在使用 GridSearchCV 时，我们仍然将数据单次划分为训练集和测试集，这可能会导致结果不稳定，也让我们过于依赖数据的此次划分。我们可以再深入一点，不是只将原始数据一次划分为训练集和测试集，而是使用交叉验证进行多次划分，这就是所谓的**嵌套交叉验证**（nested cross-validation）。在嵌套交叉验证中，有一个外层循环，遍历将数据划分为训练集和测试集的所有划分。对于每种划分都运行一次网格搜索（对于外层循环的每种划分可能会得到不同的最佳参数）。然后，对于每种外层划分，利用最佳参数设置计算得到测试集分数。

这一过程的结果是由分数组成的列表——不是一个模型，也不是一种参数设置。这些分数告诉我们在网格找到的最佳参数下模型的泛化能力好坏。由于嵌套交叉验证不提供可用于新数据的模型，所以在寻找可用于未来数据的预测模型时很少用到它。但是，它对于评估给定模型在特定数据集上的效果很有用。

在 scikit-learn 中实现嵌套交叉验证很简单。我们调用 cross_val_score，并用 GridSearchCV 的一个实例作为模型：

In[34]:

```
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                         iris.data, iris.target, cv=5)
print("Cross-validation scores: ", scores)
print("Mean cross-validation score: ", scores.mean())
```

Out[34]:

```
Cross-validation scores:  [ 0.967  1.     0.967  0.967  1.   ]
Mean cross-validation score:  0.98
```

嵌套交叉验证的结果可以总结为“SVC 在 iris 数据集上的交叉验证平均精度为 98%”——不多也不少。

这里我们在内层循环和外层循环中都使用了分层 5 折交叉验证。由于 param_grid 包含 36 种参数组合，所以需要构建 $36 \times 5 \times 5 = 900$ 个模型，导致嵌套交叉验证过程的代价很高。这里我们在内层循环和外层循环中使用相同的交叉验证分离器，但这不是必需的，你可以在内层循环和外层循环中使用交叉验证策略的任意组合。理解上面单行代码的内容可能有点困难，将其展开为 for 循环可能会有所帮助，正如我们在下面这个简化的实现中所做的那样：

In[35]:

```
def nested_cv(X, y, inner_cv, outer_cv, Classifier, parameter_grid):
    outer_scores = []
    # 对于外层交叉验证的每次数据划分，split方法返回索引值
    for training_samples, test_samples in outer_cv.split(X, y):
        # 利用内层交叉验证找到最佳参数
        best_parms = {}
        best_score = -np.inf
        # 遍历参数
        for parameters in parameter_grid:
            # 在内层划分中累加分数
            cv_scores = []
            # 遍历内层交叉验证
            for inner_train, inner_test in inner_cv.split(
                    X[training_samples], y[training_samples]):
                # 对于给定的参数和训练数据来构建分类器
                clf = Classifier(**parameters)
                clf.fit(X[inner_train], y[inner_train])
                # 在内层测试集上进行评估
                score = clf.score(X[inner_test], y[inner_test])
                cv_scores.append(score)
            # 计算内层交叉验证的平均分数
            mean_score = np.mean(cv_scores)
            if mean_score > best_score:
                # 如果比前面的模型都要好，则保存其参数
                best_score = mean_score
                best_params = parameters
        # 利用外层训练集和最佳参数来构建模型
        clf = Classifier(**best_params)
        clf.fit(X[training_samples], y[training_samples])
        # 评估模型
        outer_scores.append(clf.score(X[test_samples], y[test_samples]))
return np.array(outer_scores)
```

下面我们在 `iris` 数据集上运行这个函数：

In[36]:

```
from sklearn.model_selection import ParameterGrid, StratifiedKFold
scores = nested_cv(iris.data, iris.target, StratifiedKFold(5),
          StratifiedKFold(5), SVC, ParameterGrid(param_grid))
print("Cross-validation scores: {}".format(scores))
```

Out[36]:

```
Cross-validation scores: [ 0.967  1.     0.967  0.967  1.   ]
```

(2) 交叉验证与网格搜索并行

虽然在许多参数上运行网格搜索和在大型数据集上运行网格搜索的计算量可能很大，但令人尴尬的是，这些计算都是**并行的**（parallel）。这也就是说，在一种交叉验证划分下使用特定参数设置来构建一个模型，与利用其他参数的模型是完全独立的。这使得网格搜索与交叉验证成为多个 CPU 内核或集群上并行化的理想选择。你可以将 `n_jobs` 参数设置为你想使用的 CPU 内核数量，从而在 `GridSearchCV` 和 `cross_val_score` 中使用多个内核。你可以设置 `n_jobs=-1` 来使用所有可用的内核。

你应该知道，`scikit-learn` **不允许并行操作的嵌套**。因此，如果你在模型（比如随机森林）中使用了 `n_jobs` 选项，那么就不能在 `GridSearchCV` 使用它来搜索这个模型。如果你的数据集和模型都非常大，那么使用多个内核可能会占用大量内存，你应该在并行构建大型模型时监视内存的使用情况。

还可以在集群内的多台机器上并行运行网格搜索和交叉验证，不过在写作本书时 `scikit-learn` 还不支持这一点。但是，如果你不介意像 5.2.1 节那样编写 `for` 循环来遍历参数的话，可以使用 IPython 并行框架来进行并行网格搜索。

对于 Spark 用户，还可以使用最新开发的 `spark-sklearn` 包（https://github.com/databricks/spark-sklearn），它允许在已经建立好的 Spark 集群上运行网格搜索。

5.3 评估指标与评分

到目前为止，我们使用精度（正确分类的样本所占的比例）来评估分类性能，使用 R^2 来评估回归性能。但是，总结监督模型在给定数据集上的表现有多种方法，这两个指标只是其中两种。在实践中，这些评估指标可能不适用于你的应用。在选择模型与调参时，选择正确的指标是很重要的。

5.3.1 牢记最终目标

在选择指标时，你应该始终牢记机器学习应用的最终目标。在实践中，我们通常不仅对精确的预测感兴趣，还希望将这些预测结果用于更大的决策过程。在选择机器学习指标之前，你应该考虑应用的高级目标，这通常被称为**商业指标**（business metric）。对于一个机器学习应用，选择特定算法的结果被称为**商业影响**（business impact）。[4] 高级目标可能是避免交通事故或者减少入院人数，也可能是吸引更多的网站用户或者让用户在你的商店中花更多的钱。在选择模型或调参时，你应该选择对商业指标具有最大正面影响的模型或参数值。这通常是很难的，因为要想评估某个模型的商业影响，可能需要将它放在真实的生产环境中。

在开发的初期阶段调参，仅为了测试就将模型投入生产环境往往是不可行的，因为可能涉及很高的商业风险或个人风险。想象一下，为了测试无人驾驶汽车的行人避让能力，没有事先验证就让它直接上路。如果模型很糟糕的话，行人就会遇到麻烦！因此，我们通常需要找到某种替代的评估程序，使用一种更容易计算的评估指标。例如，我们可以测试对行人和非行人的图片进行分类并测量精度。请记住，这只是一种替代方法，找到与原始商业目标最接近的可评估的指标也很有用。应尽可能使用这个最接近的指标来进行模型评估与选择。评估的结果可能不是一个数字——算法的结果可能是顾客多了 10%，但每位顾客的花费减少了 15%——但它应该给出选择一个模型而不选另一个所造成的预期商业影响。

本节我们将首先讨论二分类这一重要特例的指标，然后转向多分类问题，最后讨论回归问题。

注 4：请具有科学头脑的读者原谅本节中出现的商业语言。不忘最终目标在科学中也同样重要，但作者想不到在这一领域中与“商业影响”具有类似含义的词语。

5.3.2　二分类指标

二分类可能是实践中最常见的机器学习应用，也是概念最简单的应用。但是，即使是评估这个简单任务也仍有一些注意事项。在深入研究替代指标之前，我们先看一下测量精度可能会如何误导我们。请记住，对于二分类问题，我们通常会说**正类**（positive class）和**反类**（negative class），而正类是我们要寻找的类。

1. 错误类型

通常来说，精度并不能很好地度量预测性能，因为我们所犯错误的数量并不包含我们感兴趣的所有信息。想象一个应用，利用自动化测试来筛查癌症的早期发现。如果测试结果为阴性，那么认为患者是健康的，而如果测试结果为阳性，患者则需要接受额外的筛查。这里我们将阳性测试结果（表示患有癌症）称为正类，将阴性测试结果称为反类。我们不能假设模型永远是完美的，它也会犯错。对于任何应用而言，我们都需要问问自己，这些错误在现实世界中可能有什么后果。

一种可能的错误是健康的患者被诊断为阳性，导致需要进行额外的测试。这给患者带来了一些费用支出和不便（可能还有精神上的痛苦）。错误的阳性预测叫作**假正例**（false positive）。另一种可能的错误是患病的人被诊断为阴性，因而不会接受进一步的检查和治疗。未诊断出的癌症可能导致严重的健康问题，甚至可能致命。这种类型的错误（错误的阴性预测）叫作**假反例**（false negative）。在统计学中，假正例也叫作**第一类错误**（type I error），假反例也叫作**第二类错误**（type II error）。我们将坚持使用“假正例”和“假反例”的说法，因为它们的含义更加明确，也更好记。在癌症诊断的例子中，显然，我们希望尽量避免假反例，而假正例可以被看作是小麻烦。

虽然这是一个特别极端的例子，但假正例和假反例造成的结果很少相同。在商业应用中，可以为两种类型的错误分配美元值，即用美元而不是精度来度量某个预测结果的错误。对于选择使用哪种模型的商业决策而言，这种方法可能更有意义。

2. 不平衡数据集

如果在两个类别中，一个类别的出现次数比另一个多很多，那么错误类型将发挥重要作用。这在实践中十分常见，一个很好的例子是点击（click-through）预测，其中每个数据点表示一个“印象”（impression），即向用户展示的一个物项。这个物项可能是广告、相关的故事，或者是在社交媒体网站上关注的相关人员。目标是预测用户是否会点击看到的某个特定物项（表示他们感兴趣）。用户对互联网上显示的大多数内容（尤其是广告）都不会点击。你可能需要向用户展示 100 个广告或文章，他们才会找到足够有趣的内容来点击查看。这样就会得到一个数据集，其中每 99 个“未点击”的数据点才有 1 个“已点击”的数据点。换句话说，99% 的样本属于“未点击”类别。这种一个类别比另一个类别出现次数多很多的数据集，通常叫作**不平衡数据集**（imbalanced dataset）或者**具有不平衡类别的数据集**（dataset with imbalanced classes）。在实际当中，不平衡数据才是常态，而数据中感兴趣事件的出现次数相同或相似的情况十分罕见。

现在假设你在构建了一个在点击预测任务中精度达到 99% 的分类器。这告诉了你什么？99% 的精度听起来令人印象深刻，但是它并没有考虑类别不平衡。你不必构建机器学习模型，始终预测“未点击”就可以得到 99% 的精度。另一方面，即使是不平衡数据，精度达

到 99% 的模型实际上也是相当不错的。但是，精度无法帮助我们区分不变的“未点击”模型与潜在的优秀模型。

为了便于说明，我们将 digits 数据集中的数字 9 与其他九个类别加以区分，从而创建一个 9:1 的不平衡数据集：

In[37]:

```
from sklearn.datasets import load_digits

digits = load_digits()
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)
```

我们可以使用 DummyClassifier 来始终预测多数类（这里是“非 9”），以查看精度提供的信息量有多么少：

In[38]:

```
from sklearn.dummy import DummyClassifier
dummy_majority = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
pred_most_frequent = dummy_majority.predict(X_test)
print("Unique predicted labels: {}".format(np.unique(pred_most_frequent)))
print("Test score: {:.2f}".format(dummy_majority.score(X_test, y_test)))
```

Out[38]:

```
Unique predicted labels: [False]
Test score: 0.90
```

我们得到了接近 90% 的精度，却没有学到任何内容。这个结果可能看起来相当好，但请思考一会儿。想象一下，有人告诉你他们的模型精度达到 90%。你可能会认为他们做得很好。但根据具体问题，也可能是仅预测了一个类别！我们将这个结果与使用一个真实分类器的结果进行对比：

In[39]:

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
pred_tree = tree.predict(X_test)
print("Test score: {:.2f}".format(tree.score(X_test, y_test)))
```

Out[39]:

```
Test score: 0.92
```

从精度来看，DecisionTreeClassifier 仅比常数预测稍好一点。这可能表示我们使用 DecisionTreeClassifier 的方法有误，也可能是因为精度实际上在这里不是一个很好的度量。

为了便于对比，我们再评估两个分类器，LogisticRegression 与默认的 DummyClassifier，其中后者进行随机预测，但预测类别的比例与训练集中的比例相同 [5]：

注 5：由于此处 DummyClassifier 未设置 random_state，因此实际运行结果可能和书中不同。——译者注

In[40]:

```
from sklearn.linear_model import LogisticRegression

dummy = DummyClassifier().fit(X_train, y_train)
pred_dummy = dummy.predict(X_test)
print("dummy score: {:.2f}".format(dummy.score(X_test, y_test)))

logreg = LogisticRegression(C=0.1).fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)
print("logreg score: {:.2f}".format(logreg.score(X_test, y_test)))
```

Out[40]:

```
dummy score: 0.80
logreg score: 0.98
```

显而易见，产生随机输出的虚拟分类器是所有分类器中最差的（精度最低），而 `LogisticRegression` 则给出了非常好的结果。但是，即使是随机分类器也得到了超过 80% 的精度。这样很难判断哪些结果是真正有帮助的。这里的问题在于，要想对这种不平衡数据的预测性能进行量化，精度并不是一种合适的度量。在本章接下来的内容中，我们将探索在选择模型方面能够提供更好指导的其他指标。我们特别希望有一个指标可以告诉我们，一个模型比“最常见”预测（由 `pred_most_frequent` 给出）或随机预测（由 `pred_dummy` 给出）要好多少。如果我们用一个指标来评估模型，那么这个指标应该能够淘汰这些无意义的预测。

3. 混淆矩阵

对于二分类问题的评估结果，一种最全面的表示方法是使用混淆矩阵（confusion matrix）。我们利用 `confusion_matrix` 函数来检查上一节中 `LogisticRegression` 的预测结果。我们已经将测试集上的预测结果保存在 `pred_logreg` 中：

In[41]:

```
from sklearn.metrics import confusion_matrix

confusion = confusion_matrix(y_test, pred_logreg)
print("Confusion matrix:\n{}".format(confusion))
```

Out[41]:

```
Confusion matrix:
[[401   2]
 [  8  39]]
```

`confusion_matrix` 的输出是一个 2×2 数组，其中行对应于真实的类别，列对应于预测的类别。数组中每个元素给出属于该行对应类别（这里是“非 9”和“9”）的样本被分类到该列对应类别中的数量。图 5-10 对这一含义进行了说明。

In[42]:

```
mglearn.plots.plot_confusion_matrix_illustration()
```

混淆矩阵主对角线 [6] 上的元素对应于正确的分类，而其他元素则告诉我们一个类别中有多少样本被错误地划分到其他类别中。

注 6：对于一个二维数组或矩阵 A，其主对角线是 A[i, i]。

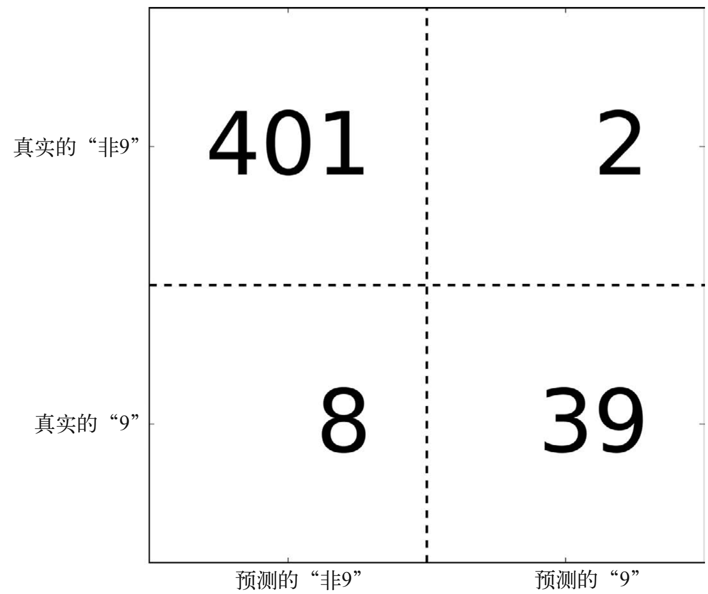


图 5-10：“9 与其他”分类任务的混淆矩阵

如果我们将“9”作为正类，那么就可以将混淆矩阵的元素与前面介绍过的**假正例**（false positive）和**假反例**（false negative）两个术语联系起来。为了使图像更加完整，我们将正类中正确分类的样本称为**真正例**（true positive），将反类中正确分类的样本称为**真反例**（true negative）。这些术语通常缩写为 FP、FN、TP 和 TN，这样就可以得到下图对混淆矩阵的解释（图 5-11）：

In[43]:

```
mglearn.plots.plot_binary_confusion_matrix()
```

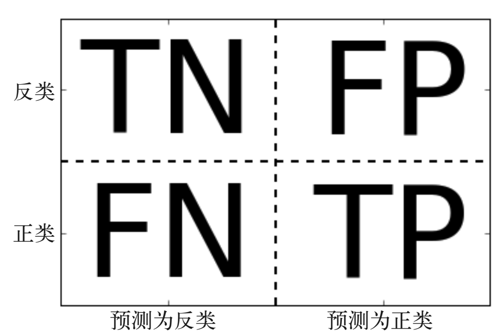


图 5-11：二分类混淆矩阵

下面我们用混淆矩阵来比较前面拟合过的模型（两个虚拟模型、决策树和 Logistic 回归）：

In[44]:

```
print("Most frequent class:")
print(confusion_matrix(y_test, pred_most_frequent))
print("\nDummy model:")
print(confusion_matrix(y_test, pred_dummy))
print("\nDecision tree:")
print(confusion_matrix(y_test, pred_tree))
print("\nLogistic Regression")
print(confusion_matrix(y_test, pred_logreg))
```

Out[44]:

```
Most frequent class:
[[403   0]
 [ 47   0]]

Dummy model:
[[361  42]
 [ 43   4]]

Decision tree:
[[390  13]
 [ 24  23]]

Logistic Regression
[[401   2]
 [  8  39]]
```

观察混淆矩阵，很明显可以看出 `pred_most_frequent` 有问题，因为它总是预测同一个类别。另一方面，`pred_dummy` 的真正例数量很少（4 个），特别是与假反例和假正例的数量相比——假正例的数量竟然比真正例还多！决策树的预测比虚拟预测更有意义，即使二者精度几乎相同。最后，我们可以看到，Logistic 回归在各方面都比 `pred_tree` 要好：它的真正例和真反例的数量更多，而假正例和假反例的数量更少。从这个对比中可以明确看出，只有决策树和 Logistic 回归给出了合理的结果，并且 Logistic 回归的效果全面好于决策树。但是，检查整个混淆矩阵有点麻烦，虽然我们通过观察矩阵的各个方面得到了很多深入见解，但是这个过程是人工完成的，也是非常定性的。有几种方法可以总结混淆矩阵中包含的信息，我们将在后面进行讨论。

与精度的关系。我们已经讲过一种总结混淆矩阵结果的方法——计算精度，其公式表达如下所示：

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

换句话说，精度是正确预测的数量（TP 和 TN）除以所有样本的数量（混淆矩阵中所有元素的总和）。

准确率、召回率与 *f*-分数。总结混淆矩阵还有几种方法，其中最常见的就是准确率和召回率。**准确率**（precision）度量的是被预测为正例的样本中有多少是真正的正例：

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

如果目标是限制假正例的数量，那么可以使用准确率作为性能指标。举个例子，想象一个模型，它预测一种新药在临床试验治疗中是否有效。众所周知，临床试验非常昂贵，制药公司只有在非常确定药物有效的情况下才会进行试验。因此，模型不会产生很多假正例是很重要的——换句话说，模型的准确率很高。准确率也被称为**阳性预测值**（positive predictive value，PPV）。

另一方面，**召回率**（recall）度量的是正类样本中有多少被预测为正类：

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

如果我们需要找出所有的正类样本，即避免假反例是很重要的情况下，那么可以使用召回率作为性能指标。本章前面的癌症诊断例子就是一个很好的例子：找出所有患病的人很重要，预测结果中可能包含健康的人。召回率的其他名称有**灵敏度**（sensitivity）、**命中率**（hit rate）和**真正例率**（true positive rate，TPR）。

在优化召回率与优化准确率之间需要折中。如果你预测所有样本都属于正类，那么可以轻松得到完美的召回率——没有假反例，也没有真反例。但是，将所有样本都预测为正类，将会得到许多假正例，因此准确率会很低。与之相反，如果你的模型只将一个最确定的数据点预测为正类，其他点都预测为反类，那么准确率将会很完美（假设这个数据点实际上就属于正类），但是召回率会非常差。

> 准确率和召回率只是从 TP、FP、TN 和 FN 导出的众多分类度量中的两个。你可以在 Wikipedia 上找到所有度量的摘要（https://en.wikipedia.org/wiki/Sensitivity_and_specificity）。在机器学习社区中，准确率和召回率是最常用的二分类度量，但其他社区可能使用其他相关指标。

虽然准确率和召回率是非常重要的度量，但是仅查看二者之一无法为你提供完整的图景。将两种度量进行汇总的一种方法是 ***f*-分数**（*f*-score）或 ***f*-度量**（*f*-measure），它是准确率与召回率的调和平均：

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

这一特定变体也被称为 **f_1-分数**（f_1-score）。由于同时考虑了准确率和召回率，所以它对于不平衡的二分类数据集来说是一种比精度更好的度量。我们对之前计算过的“9 与其余”数据集的预测结果计算 f_1- 分数。这里我们假定“9”类是正类（标记为 `True`，其他样本被标记为 `False`），因此正类是少数类：

In[45]:

```
from sklearn.metrics import f1_score
print("f1 score most frequent: {:.2f}".format(
        f1_score(y_test, pred_most_frequent)))
```

```
print("f1 score dummy: {:.2f}".format(f1_score(y_test, pred_dummy)))
print("f1 score tree: {:.2f}".format(f1_score(y_test, pred_tree)))
print("f1 score logistic regression: {:.2f}".format(
        f1_score(y_test, pred_logreg)))
```

Out[45]:

```
f1 score most frequent: 0.00
f1 score dummy: 0.10
f1 score tree: 0.55
f1 score logistic regression: 0.89
```

这里我们可以注意到两件事情。第一，我们从 `most_frequent` 的预测中得到一条错误信息，因为预测的正类数量为 0（使得 *f*- 分数的分母为 0）。第二，我们可以看到虚拟预测与决策树预测之间有很大的区别，而仅观察精度时二者的区别并不明显。利用 *f*- 分数进行评估，我们再次用一个数字总结了预测性能。但是，*f*- 分数似乎比精度更加符合我们对好模型的直觉。然而，*f*- 分数的一个缺点是比精度更加难以解释。

如果我们想要对准确率、召回率和 f_1- 分数做一个更全面的总结，可以使用 `classification_report` 这个很方便的函数，它可以同时计算这三个值，并以美观的格式打印出来：

In[46]:

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred_most_frequent,
                            target_names=["not nine", "nine"]))
```

Out[46]:

```
             precision    recall  f1-score   support

   not nine       0.90      1.00      0.94       403
       nine       0.00      0.00      0.00        47

avg / total       0.80      0.90      0.85       450
```

`classification_report` 函数为每个类别（这里是 `True` 和 `False`）生成一行，并给出以该类别作为正类的准确率、召回率和 *f*- 分数。前面我们假设较少的"9"类是正类。如果将正类改为"not nine"（非 9），我们可以从 `classification_report` 的输出中看出，利用 `most_frequent` 模型得到的 *f*- 分数为 0.94。此外，对于"not nine"类别，召回率是 1，因为我们将所有样本都分类为"not nine"。*f*- 分数旁边的最后一列给出了每个类别的**支持**（support），它表示的是在这个类别中真实样本的数量。

分类报告的最后一行显示的是对应指标的加权平均（按每个类别中的样本个数加权）。下面还有两个报告，一个是虚拟分类器的 [7]，一个是 Logistic 回归的：

In[47]:

```
print(classification_report(y_test, pred_dummy,
                            target_names=["not nine", "nine"]))
```

注 7：由于前面 In[40] 中的 `DummyClassifier` 未设置 `random_state`，因此此处的实际运行结果可能与书中不同。——译者注

Out[47]:

```
              precision    recall  f1-score   support

    not nine       0.90      0.92      0.91       403
        nine       0.11      0.09      0.10        47

 avg / total       0.81      0.83      0.82       450
```

In[48]:

```
print(classification_report(y_test, pred_logreg,
                            target_names=["not nine", "nine"]))
```

Out[48]:

```
              precision    recall  f1-score   support

    not nine       0.98      1.00      0.99       403
        nine       0.95      0.83      0.89        47

 avg / total       0.98      0.98      0.98       450
```

在查看报告时你可能注意到了，虚拟模型与好模型之间的区别不再那么明显。选择哪个类作为正类对指标有很大影响。虽然在以“nine”类作为正类时虚拟分类的 *f*- 分数是 0.10（对比 Logistic 回归的 0.89），而以“not nine”类作为正类时二者的 *f*- 分数分别是 0.91 和 0.99，两个结果看起来都很合理。不过同时查看所有数字可以给出非常准确的图像，我们可以清楚地看到 Logistic 回归模型的优势。

4. 考虑不确定性

混淆矩阵和分类报告为一组特定的预测提供了非常详细的分析。但是，预测本身已经丢弃了模型中包含的大量信息。正如我们在第 2 章中所讨论的那样，大多数分类器都提供了一个 `decision_function` 或 `predict_proba` 方法来评估预测的不确定度。预测可以被看作是以某个固定点作为 `decision_function` 或 `predict_proba` 输出的阈值——在二分类问题中，我们使用 0 作为决策函数的阈值，0.5 作为 `predict_proba` 的阈值。

下面是一个不平衡二分类任务的示例，反类中有 400 个点，而正类中只有 50 个点。训练数据如图 5-12 左侧所示。我们在这个数据上训练一个核 SVM 模型，训练数据右侧的图像将决策函数值绘制为热图。你可以在图像偏上的位置看到一个黑色圆圈，表示 `decision_function` 的阈值刚好为 0。在这个圆圈内的点将被划为正类，圆圈外的点将被划为反类：

In[49]:

```
from mglearn.datasets import make_blobs
X, y = make_blobs(n_samples=(400, 50), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
```

In[50]:

```
mglearn.plots.plot_decision_threshold()
```

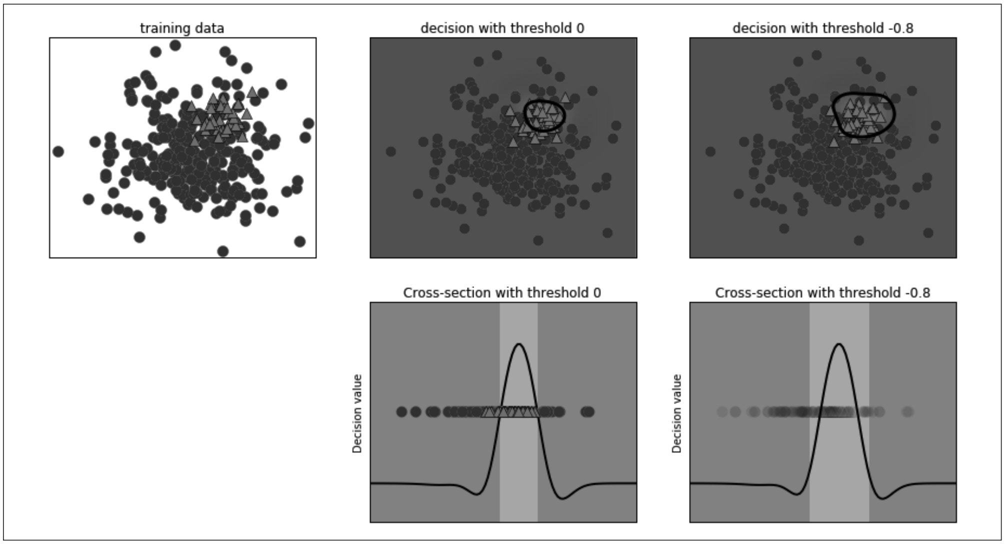


图 5-12：决策函数的热图与改变决策阈值的影响

我们可以使用 classification_report 函数来评估两个类别的准确率与召回率：

In[51]:

```
print(classification_report(y_test, svc.predict(X_test)))
```

Out[51]:

```
             precision    recall  f1-score   support

          0       0.97      0.89      0.93       104
          1       0.35      0.67      0.46         9

avg / total       0.92      0.88      0.89       113
```

对于类别 1，我们得到了一个相当低的准确率，而召回率则令人糊涂（mixed）。由于类别 0 要大得多，所以分类器将重点放在将类别 0 分类正确，而不是较小的类别 1。

假设在我们的应用中，类别 1 具有高召回率更加重要，正如前面的癌症筛查例子那样。这意味着我们愿意冒险有更多的假正例（假的类别 1），以换取更多的真正例（可增大召回率）。svc.predict 生成的预测无法满足这个要求，但我们可以通过改变决策阈值不等于 0 来将预测重点放在使类别 1 的召回率更高。默认情况下，decision_function 值大于 0 的点将被划为类别 1。我们希望将**更多**的点划为类别 1，所以需要**减小**阈值：

In[52]:

```
y_pred_lower_threshold = svc.decision_function(X_test) > -.8
```

我们来看一下这个预测的分类报告：

In[53]:

```
print(classification_report(y_test, y_pred_lower_threshold))
```

```
Out[53]:
              precision    recall  f1-score   support

          0       1.00      0.82      0.90       104
          1       0.32      1.00      0.49         9

avg / total       0.95      0.83      0.87       113
```

正如所料，类别 1 的召回率增大，准确率减小。现在我们将更大的空间区域划为类别 1，正如图 5-12 右上图中所示。如果你认为准确率比召回率更重要，或者反过来，或者你的数据严重不平衡，那么改变决策阈值是得到更好结果的最简单方法。由于 `decision_function` 的取值可能在任意范围，所以很难提供关于如何选取阈值的经验法则。

> 如果你设置了阈值，那么要小心不要在测试集上这么做。与其他任何参数一样，在测试集上设置决策阈值可能会得到过于乐观的结果。可以使用验证集或交叉验证来代替。

对于实现了 `predict_proba` 方法的模型来说，选择阈值可能更简单，因为 `predict_proba` 的输出固定在 0 到 1 的范围内，表示的是概率。默认情况下，0.5 的阈值表示，如果模型以超过 50% 的概率"确信"一个点属于正类，那么就将其划为正类。增大这个阈值意味着模型需要更加确信才能做出正类的判断（较低程度的确信就可以做出反类的判断）。虽然使用概率可能比使用任意阈值更加直观，但并非所有模型都提供了不确定性的实际模型（一棵生长到最大深度的 `DecisionTree` 总是 100% 确信其判断，即使很可能是错的）。这与**校准**（calibration）的概念相关：校准模型是指能够为其不确定性提供精确度量的模型。校准的详细讨论超出了本书的范围，但你可以在 Alexandru Niculescu-Mizil 和 Rich Caruana 的"Predicting Good Probabilities with Supervised Learning"（http://www.machinelearning.org/proceedings/icml2005/papers/079_GoodProbabilities_NiculescuMizilCaruana.pdf）这篇文章中找到更多内容。

5. 准确率-召回率曲线

如前所述，改变模型中用于做出分类决策的阈值，是一种调节给定分类器的准确率和召回率之间折中的方法。你可能希望仅遗漏不到 10% 的正类样本，即希望召回率能达到 90%。这一决策取决于应用，应该是由商业目标驱动的。一旦设定了一个具体目标（比如对某一类别的特定召回率或准确率），就可以适当地设定一个阈值。总是可以设置一个阈值来满足特定的目标，比如 90% 的召回率。难点在于开发一个模型，在满足这个阈值的同时仍具有合理的准确率——如果你将所有样本都划为正类，那么将会得到 100% 的召回率，但你的模型毫无用处。

对分类器设置要求（比如 90% 的召回率）通常被称为设置**工作点**（operating point）。在业务中固定工作点通常有助于为客户或组织内的其他小组提供性能保证。

在开发新模型时，通常并不完全清楚工作点在哪里。因此，为了更好地理解建模问题，很有启发性的做法是，**同时**查看所有可能的阈值或准确率和召回率的所有可能折中。利用一种叫作**准确率 – 召回率曲线**（precision-recall curve）的工具可以做到这一点。你可以在 `sklearn.metrics` 模块中找到计算准确率 – 召回率曲线的函数。这个函数需要真实标签与预测的不确定度，后者由 `decision_function` 或 `predict_proba` 给出：

In[54]:

```
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
```

precision_recall_curve 函数返回一个列表，包含按顺序排序的所有可能阈值（在决策函数中出现的所有值）对应的准确率和召回率，这样我们就可以绘制一条曲线，如图 5-13 所示：

In[55]:

```
# 使用更多数据点来得到更加平滑的曲线
X, y = make_blobs(n_samples=(4000, 500), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
# 找到最接近于0的阈值
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="threshold zero", fillstyle="none", c='k', mew=2)

plt.plot(precision, recall, label="precision recall curve")
plt.xlabel("Precision")
plt.ylabel("Recall")
```

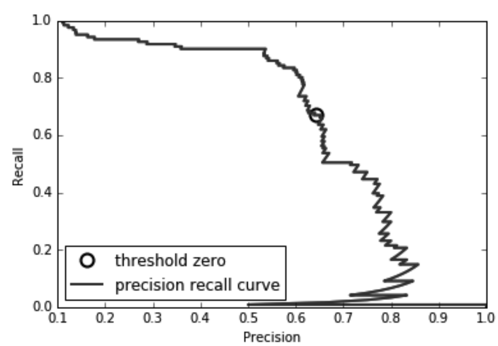


图 5-13：SVC（gamma=0.05）的准确率 – 召回率曲线

图 5-13 中曲线上的每一个点都对应 decision_function 的一个可能的阈值。例如，我们可以看到，在准确率约为 0.75 的位置对应的召回率为 0.4。黑色圆圈表示的是阈值为 0 的点，0 是 decision_function 的默认阈值。这个点是在调用 predict 方法时所选择的折中点。

曲线越靠近右上角，则分类器越好。右上角的点表示对于同一个阈值，准确率和召回率都很高。曲线从左上角开始，这里对应于非常低的阈值，将所有样本都划为正类。提高阈值可以让曲线向准确率更高的方向移动，但同时召回率降低。继续增大阈值，大多数被划为正类的点都是真正例，此时准确率很高，但召回率更低。随着准确率的升高，模型越能够保持较高的召回率，则模型越好。

进一步观察这条曲线，可以发现，利用这个模型可以得到约 0.5 的准确率，同时保持很高的召回率。如果我们想要更高的准确率，那么就必须牺牲很多召回率。换句话说，曲线左侧相对平坦，说明在准确率提高的同时召回率没有下降很多。当准确率大于 0.5 之后，准确率每增加一点都会导致召回率下降许多。

不同的分类器可能在曲线上不同的位置（即在不同的工作点）表现很好。我们来比较一下在同一数据集上训练的 SVM 与随机森林。RandomForestClassifier 没有 decision_function，只有 predict_proba。precision_recall_curve 函数的第二个参数应该是正类（类别 1）的确定性度量，所以我们传入样本属于类别 1 的概率（即 rf.predict_proba(X_test)[:, 1]）。二分类问题的 predict_proba 的默认阈值是 0.5，所以我们在曲线上标出这个点（见图 5-14）：

In[56]:

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=100, random_state=0, max_features=2)
rf.fit(X_train, y_train)

# RandomForestClassifier有predict_proba，但没有decision_function
precision_rf, recall_rf, thresholds_rf = precision_recall_curve(
    y_test, rf.predict_proba(X_test)[:, 1])

plt.plot(precision, recall, label="svc")

plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="threshold zero svc", fillstyle="none", c='k', mew=2)

plt.plot(precision_rf, recall_rf, label="rf")

close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(precision_rf[close_default_rf], recall_rf[close_default_rf], '^', c='k',
         markersize=10, label="threshold 0.5 rf", fillstyle="none", mew=2)
plt.xlabel("Precision")
plt.ylabel("Recall")
plt.legend(loc="best")
```

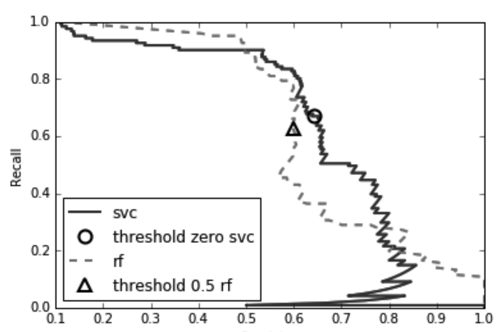


图 5-14：比较 SVM 与随机森林的准确率 – 召回率曲线

从这张对比图中可以看出，随机森林在极值处（要求很高的召回率或很高的准确率）的表现更好。在中间位置（准确率约为 0.7）SVM 的表现更好。如果我们只查看 f_1- 分数来比较二者的总体性能，那么可能会遗漏这些细节。f_1- 分数只反映了准确率 – 召回率曲线上的一个点，即默认阈值对应的那个点：

In[57]:

```
print("f1_score of random forest: {:.3f}".format(
    f1_score(y_test, rf.predict(X_test))))
print("f1_score of svc: {:.3f}".format(f1_score(y_test, svc.predict(X_test))))
```

Out[57]:

```
f1_score of random forest: 0.610
f1_score of svc: 0.656
```

比较这两条准确率 – 召回率曲线，可以为我们提供大量详细的洞见，但这是一个相当麻烦的过程。对于自动化模型对比，我们可能希望总结曲线中包含的信息，而不限于某个特定的阈值或工作点。总结准确率 – 召回率曲线的一种方法是计算该曲线下的积分或面积，也叫作**平均准确率**（average precision）。[8] 你可以使用 `average_precision_score` 函数来计算平均准确率。因为我们要计算准确率 – 召回率曲线并考虑多个阈值，所以需要向 `average_precision_score` 传入 `decision_function` 或 `predict_proba` 的结果，而不是 `predict` 的结果：

In[58]:

```
from sklearn.metrics import average_precision_score
ap_rf = average_precision_score(y_test, rf.predict_proba(X_test)[:, 1])
ap_svc = average_precision_score(y_test, svc.decision_function(X_test))
print("Average precision of random forest: {:.3f}".format(ap_rf))
print("Average precision of svc: {:.3f}".format(ap_svc))
```

Out[58]:

```
Average precision of random forest: 0.666
Average precision of svc: 0.663
```

在对所有可能的阈值进行平均时，我们看到随机森林和 SVC 的表现差不多好，随机森林稍稍领先。这与前面从 `f1_score` 中得到的结果大为不同。因为平均准确率是从 0 到 1 的曲线下的面积，所以平均准确率总是返回一个在 0（最差）到 1（最好）之间的值。随机分配 `decision_function` 的分类器的平均准确率是数据集中正例样本所占的比例。

6. 受试者工作特征（ROC）与AUC

还有一种常用的工具可以分析不同阈值的分类器行为：**受试者工作特征曲线**（receiver operating characteristics curve），简称为 **ROC 曲线**（ROC curve）。与准确率 – 召回率曲线类似，ROC 曲线考虑了给定分类器的所有可能的阈值，但它显示的是**假正例率**（false positive rate，FPR）和**真正例率**（true positive rate，TPR），而不是报告准确率和召回率。回想一下，真正例率只是召回率的另一个名称，而假正例率则是假正例占所有反类样本的比例：

注 8：准确率 – 召回率曲线下的面积与平均准确率之间还有一些较小的技术区别。但这个解释表达的是大致思路。

$$FPR = \frac{FP}{FP + TN}$$

可以用 roc_curve 函数来计算 ROC 曲线（见图 5-15）：

In[59]:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, svc.decision_function(X_test))

plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
# 找到最接近于0的阈值
close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
         label="threshold zero", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```

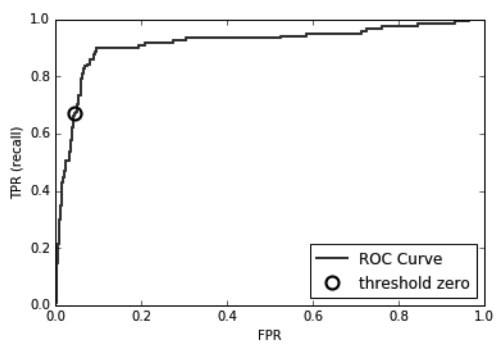


图 5-15：SVM 的 ROC 曲线

对于 ROC 曲线，理想的曲线要靠近左上角：你希望分类器的召回率很高，同时保持假正例率很低。从曲线中可以看出，与默认阈值 0 相比，我们可以得到明显更高的召回率（约 0.9），而 FPR 仅稍有增加。最接近左上角的点可能是比默认选择更好的工作点。同样请注意，不应该在测试集上选择阈值，而是应该在单独的验证集上选择。

图 5-16 给出了随机森林和 SVM 的 ROC 曲线对比：

In[60]:

```
from sklearn.metrics import roc_curve
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, rf.predict_proba(X_test)[:, 1])

plt.plot(fpr, tpr, label="ROC Curve SVC")
plt.plot(fpr_rf, tpr_rf, label="ROC Curve RF")

plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
         label="threshold zero SVC", fillstyle="none", c='k', mew=2)
```

```
close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(fpr_rf[close_default_rf], tpr[close_default_rf], '^', markersize=10,
         label="threshold 0.5 RF", fillstyle="none", c='k', mew=2)

plt.legend(loc=4)
```

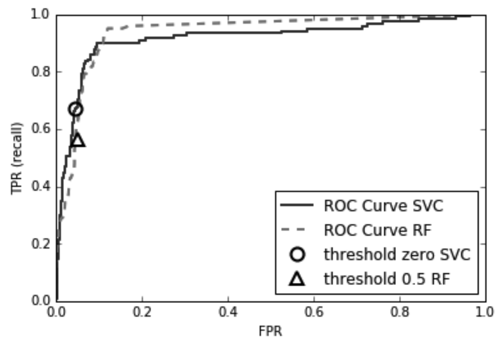


图 5-16：比较 SVM 和随机森林的 ROC 曲线

与准确率 – 召回率曲线一样，我们通常希望使用一个数字来总结 ROC 曲线，即曲线下的面积［通常被称为 AUC（area under the curve），这里的曲线指的就是 ROC 曲线］。我们可以利用 `roc_auc_score` 函数来计算 ROC 曲线下的面积：

In[61]:

```
from sklearn.metrics import roc_auc_score
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
print("AUC for Random Forest: {:.3f}".format(rf_auc))
print("AUC for SVC: {:.3f}".format(svc_auc))
```

Out[61]:

```
AUC for Random Forest: 0.937
AUC for SVC: 0.916
```

利用 AUC 分数来比较随机森林和 SVM，我们发现随机森林的表现比 SVM 要略好一些。回想一下，由于平均准确率是从 0 到 1 的曲线下的面积，所以平均准确率总是返回一个 0（最差）到 1（最好）之间的值。随机预测得到的 AUC 总是等于 0.5，无论数据集中的类别多么不平衡。对于不平衡的分类问题来说，AUC 是一个比精度好得多的指标。AUC 可以被解释为评估正例样本的**排名**（ranking）。它等价于从正类样本中随机挑选一个点，由分类器给出的分数比从反类样本中随机挑选一个点的分数更高的概率。因此，AUC 最高为 1，这说明所有正类点的分数高于所有反类点。对于不平衡类别的分类问题，使用 AUC 进行模型选择通常比使用精度更有意义。

我们回到前面研究过的例子：将 `digits` 数据集中的所有 9 与所有其他数据加以区分。我们将使用 SVM 对数据集进行分类，分别使用三种不同的内核宽度（`gamma`）设置（参见图 5-17）：

In[62]:

```
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)

plt.figure()

for gamma in [1, 0.05, 0.01]:
    svc = SVC(gamma=gamma).fit(X_train, y_train)
    accuracy = svc.score(X_test, y_test)
    auc = roc_auc_score(y_test, svc.decision_function(X_test))
    fpr, tpr, _ = roc_curve(y_test , svc.decision_function(X_test))
    print("gamma = {:.2f}  accuracy = {:.2f}  AUC = {:.2f}".format(
    gamma, accuracy, auc))
    plt.plot(fpr, tpr, label="gamma={:.3f}".format(gamma))
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.xlim(-0.01, 1)
plt.ylim(0, 1.02)
plt.legend(loc="best")
```

Out[62]:

```
gamma = 1.00  accuracy = 0.90  AUC = 0.50
gamma = 0.05  accuracy = 0.90  AUC = 0.90
gamma = 0.01  accuracy = 0.90  AUC = 1.00
```

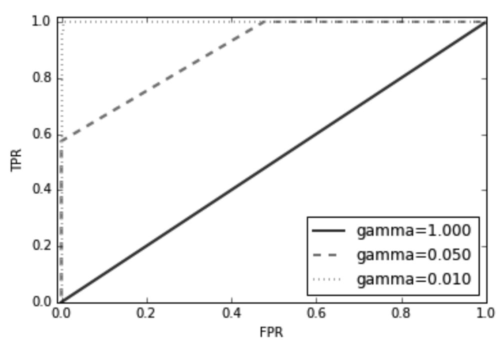


图 5-17：对比不同 gamma 值的 SVM 的 ROC 曲线[9]

对于三种不同的 gamma 设置，其精度是相同的，都等于 90%。这可能与随机选择的性能相同，也可能不同。但是观察 AUC 以及对应的曲线，我们可以看到三个模型之间有明显的区别。对于 `gamma=1.0`，AUC 实际上处于随机水平，即 `decision_function` 的输出与随机结果一样好。对于 `gamma=0.05`，性能大幅提升至 AUC 等于 0.9。最后，对于 `gamma=0.01`，

注 9：图 5-17 与 GitHub 仓库中的图像不一致（见 https://github.com/amueller/introduction_to_ml_with_python/blob/master/05-model-evaluation-and-improvement.ipynb 的 Out[65]），译者得到的图像与 GitHub 仓库中的一致。但如果用 GitHub 的这张图，下面的解释文字又对不上了。——译者注

我们得到等于 1.0 的完美 AUC。这意味着根据决策函数，所有正类点的排名要高于所有反类点。换句话说，利用正确的阈值，这个模型可以对所有数据进行完美分类！[10] 知道这一点，我们可以调节这个模型的阈值并得到很好的预测结果。如果我们仅使用精度，那么将永远不会发现这一点。

因此，我们强烈建议在不平衡数据上评估模型时使用 AUC。但请记住，AUC 没有使用默认阈值，因此，为了从高 AUC 的模型中得到有用的分类结果，可能还需要调节决策阈值。

5.3.3 多分类指标

前面我们已经深入讨论了二分类任务的评估，下面来看一下对多分类问题的评估指标。多分类问题的所有指标基本上都来自于二分类指标，但是要对所有类别进行平均。多分类的精度被定义为正确分类的样本所占的比例。同样，如果类别是不平衡的，精度并不是很好的评估度量。想象一个三分类问题，其中 85% 的数据点属于类别 A，10% 属于类别 B，5% 属于类别 C。在这个数据集上 85% 的精度说明了什么？一般来说，多分类结果比二分类结果更加难以理解。除了精度，常用的工具有混淆矩阵和分类报告，我们在上一节二分类的例子中都见过。下面我们将这两种详细的评估方法应用于对 `digits` 数据集中 10 种不同的手写数字进行分类的任务：

In[63]:

```
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
print("Accuracy: {:.3f}".format(accuracy_score(y_test, pred)))
print("Confusion matrix:\n{}".format(confusion_matrix(y_test, pred)))
```

Out[63]:

```
Accuracy: 0.953
Confusion matrix:
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  1  0  0 45  0  0]
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]
```

模型的精度为 95.3%，这表示我们已经做得相当好了。混淆矩阵为我们提供了更多细节。与二分类的情况相同，每一行对应于真实标签，每一列对应于预测标签。图 5-18 给出了一张视觉上更加吸引人的图像：

注 10：仔细观察 `gamma=0.01` 的曲线，你可以看到左上角有一个很小的弯曲。这说明至少有一个点的排名是错的。AUC 等于 1.0 是舍入到第二位小数的结果。

In[64]:

```
scores_image = mglearn.tools.heatmap(
    confusion_matrix(y_test, pred), xlabel='Predicted label',
    ylabel='True label', xticklabels=digits.target_names,
    yticklabels=digits.target_names, cmap=plt.cm.gray_r, fmt="%d")
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```

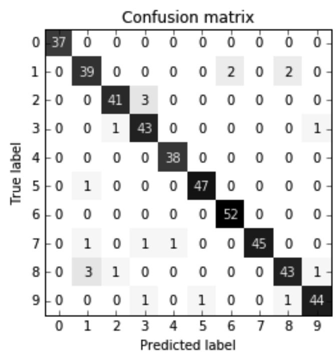


图 5-18：10 个数字分类任务的混淆矩阵

对于第一个类别（数字 0），它包含 37 个样本，所有这些样本都被划为类别 0（即类别 0 没有假反例）。我们之所以可以看出这一点，是因为混淆矩阵第一行中其他所有元素都为 0。我们还可以看到，没有其他数字被误分类为类别 0，这是因为混淆矩阵第一列中其他所有元素都为 0（即类别 0 没有假正例）。但是有些数字与其他数字混在一起——比如数字 2（第 3 行），其中有 3 个被划分到数字 3 中（第 4 列）。还有一个数字 3 被划分到数字 2 中（第 4 行第 3 列），一个数字 8 被划分到数字 2 中（第 9 行第 3 列）。

利用 `classification_report` 函数，我们可以计算每个类别的准确率、召回率和 *f*- 分数：

In[65]:

```
print(classification_report(y_test, pred))
```

Out[65]:

```
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        37
          1       0.89      0.91      0.90        43
          2       0.95      0.93      0.94        44
          3       0.90      0.96      0.92        45
          4       0.97      1.00      0.99        38
          5       0.98      0.98      0.98        48
          6       0.96      1.00      0.98        52
          7       1.00      0.94      0.97        48
          8       0.93      0.90      0.91        48
          9       0.96      0.94      0.95        47

avg / total       0.95      0.95      0.95       450
```

不出所料，类别 0 的准确率和召回率都是完美的 1，因为这个类别中没有混淆。另一方面，对于类别 7，准确率为 1，这是因为没有其他类别被误分类为 7；而类别 6 没有假反例，所以召回率等于 1。我们还可以看到，模型对类别 8 和类别 3 的表现特别不好。

对于多分类问题中的不平衡数据集，最常用的指标就是多分类版本的 *f*- 分数。多分类 *f*- 分数背后的想法是，对每个类别计算一个二分类 *f*- 分数，其中该类别是正类，其他所有类别组成反类。然后，使用以下策略之一对这些按类别 *f*- 分数进行平均。

- “宏”（macro）平均：计算未加权的按类别 *f*- 分数。它对所有类别给出相同的权重，无论类别中的样本量大小。
- “加权”（weighted）平均：以每个类别的支持作为权重来计算按类别 *f*- 分数的平均值。分类报告中给出的就是这个值。
- “微”（micro）平均：计算所有类别中假正例、假反例和真正例的总数，然后利用这些计数来计算准确率、召回率和 *f*- 分数。

如果你对每个**样本**等同看待，那么推荐使用“微”平均 f_1- 分数；如果你对每个**类别**等同看待，那么推荐使用“宏”平均 f_1- 分数：

In[66]:

```
print("Micro average f1 score: {:.3f}".format
      (f1_score(y_test, pred, average="micro")))
print("Macro average f1 score: {:.3f}".format
      (f1_score(y_test, pred, average="macro")))
```

Out[66]:

```
Micro average f1 score: 0.953
Macro average f1 score: 0.954
```

5.3.4　回归指标

对回归问题可以像分类问题一样进行详细评估，例如，对目标值估计过高与目标值估计过低进行对比分析。但是，对于我们见过的大多数应用来说，使用默认 R^2 就足够了，它由所有回归器的 `score` 方法给出。业务决策有时是根据均方误差或平均绝对误差做出的，这可能会鼓励人们使用这些指标来调节模型。但是一般来说，我们认为 R^2 是评估回归模型的更直观的指标。

5.3.5　在模型选择中使用评估指标

前面详细讨论了许多种评估方法，以及如何根据真实情况和具体模型来应用这些方法。但我们通常希望，在使用 `GridSearchCV` 或 `cross_val_score` 进行模型选择时能够使用 AUC 等指标。幸运的是，`scikit-learn` 提供了一种非常简单的实现方法，就是 `scoring` 参数，它可以同时用于 `GridSearchCV` 和 `cross_val_score`。你只需提供一个字符串，用于描述想要使用的评估指标。举个例子，我们想用 AUC 分数对 `digits` 数据集中“9 与其他”任务上的 SVM 分类器进行评估。想要将分数从默认值（精度）修改为 AUC，可以提供 `"roc_auc"` 作为 `scoring` 参数的值：

In[67]:

```
# 分类问题的默认评分是精度
print("Default scoring: {}".format(
    cross_val_score(SVC(), digits.data, digits.target == 9)))
# 指定"scoring="accuracy"不会改变结果
explicit_accuracy =  cross_val_score(SVC(), digits.data, digits.target == 9,
                                     scoring="accuracy")
print("Explicit accuracy scoring: {}".format(explicit_accuracy))
roc_auc =  cross_val_score(SVC(), digits.data, digits.target == 9,
                           scoring="roc_auc")
print("AUC scoring: {}".format(roc_auc))
```

Out[67]:

```
Default scoring: [ 0.9  0.9  0.9]
Explicit accuracy scoring: [ 0.9  0.9  0.9]
AUC scoring: [ 0.994  0.99   0.996]
```

类似地，我们可以改变 `GridSearchCV` 中用于选择最佳参数的指标：

In[68]:

```
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target == 9, random_state=0)

# 我们给出了不太好的网格来说明：
param_grid = {'gamma': [0.0001, 0.01, 0.1, 1, 10]}
# 使用默认的精度：
grid = GridSearchCV(SVC(), param_grid=param_grid)
grid.fit(X_train, y_train)
print("Grid-Search with accuracy")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (accuracy)): {:.3f}".format(grid.best_score_))
print("Test set AUC: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Test set accuracy: {:.3f}".format(grid.score(X_test, y_test)))
```

Out[68]:

```
Grid-Search with accuracy
Best parameters: {'gamma': 0.0001}
Best cross-validation score (accuracy)): 0.970
Test set AUC: 0.992
Test set accuracy: 0.973
```

In[69]:

```
# 使用AUC评分来代替：
grid = GridSearchCV(SVC(), param_grid=param_grid, scoring="roc_auc")
grid.fit(X_train, y_train)
print("\nGrid-Search with AUC")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (AUC): {:.3f}".format(grid.best_score_))
print("Test set AUC: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Test set accuracy: {:.3f}".format(grid.score(X_test, y_test)))
```

Out[69]:

```
Grid-Search with AUC
Best parameters: {'gamma': 0.01}
Best cross-validation score (AUC): 0.997
Test set AUC: 1.000
Test set accuracy: 1.000
```

在使用精度时，选择的参数是 `gamma=0.0001`，而使用 AUC 时选择的参数是 `gamma=0.01`。在两种情况下，交叉验证精度与测试集精度是一致的。但是，使用 AUC 找到的参数设置，对应的 AUC 更高，甚至对应的精度也更高。[11]

对于分类问题，`scoring` 参数最重要的取值包括：`accuracy`（默认值）、`roc_auc`（ROC 曲线下方的面积）、`average_precision`（准确率 – 召回率曲线下方的面积）、`f1`、`f1_macro`、`f1_micro` 和 `f1_weighted`（这四个是二分类的 f_1- 分数以及各种加权变体）。对于回归问题，最常用的取值包括：`r2`（R^2 分数）、`mean_squared_error`（均方误差）和 `mean_absolute_error`（平均绝对误差）。你可以在文档中找到所支持参数的完整列表（http://scikit-learn.org/stable/modules/model_evaluation.html#the-scoring-parameter-defining-model-evaluation-rules），也可以查看 `metrics.scorer` 模块中定义的 `SCORER` 字典。

In[70]:

```
from sklearn.metrics.scorer import SCORERS
print("Available scorers:\n{}".format(sorted(SCORERS.keys())))
```

Out[70]:

```
Available scorers:
['accuracy', 'adjusted_rand_score', 'average_precision', 'f1', 'f1_macro',
 'f1_micro', 'f1_samples', 'f1_weighted', 'log_loss', 'mean_absolute_error',
 'mean_squared_error', 'median_absolute_error', 'precision', 'precision_macro',
 'precision_micro', 'precision_samples', 'precision_weighted', 'r2', 'recall',
 'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc']
```

5.4　小结与展望

本章讨论了交叉验证、网格搜索和评估指标等内容，它们是评估与改进机器学习算法的基础。本章介绍的工具，以及第 2、3 章介绍的算法，是每位机器学习从业者赖以生存的工具。

本章有两个特别的要点，这里需要重复一下，因为它们经常被新的从业人员所忽视。第一个要点与交叉验证有关。交叉验证或者使用测试集让我们可以评估一个机器学习模型未来的表现。但是，如果我们使用测试集或交叉验证来选择模型或选择模型参数，那么我们就**“用完了”**测试数据，而使用相同的数据来评估模型未来的表现将会得到过于乐观的估计。因此，我们需要将数据集划分为训练数据、验证数据与测试数据，其中训练数据用于模型构建，验证数据用于选择模型与参数，测试数据用于模型评估。我们可以用交叉验证来代

注 11：利用 AUC 找到了精度更高的模型，这可能是因为对于不平衡数据来说，精度并不是模型性能的良好度量。

替每一次简单的划分。最常用的形式（如前所述）是训练 / 测试划分用于评估，然后对训练集使用交叉验证来选择模型与参数。

第二个要点与用于模型选择与模型评估的评估指标或评分函数有关。如何利用机器学习模型的预测结果做出商业决策，其理论有些超出了本书范围。[12] 但是，机器学习任务的最终目标很少是构建一个高精度的模型。一定要确保你用于模型评估与选择的指标能够很好地替代模型的实际用途。在实际当中，分类问题很少会遇到平衡的类别，假正例和假反例也通常具有非常不同的后果。你一定要了解这些后果，并选择相应的评估指标。

到目前为止，我们介绍的模型评估与选择技术都是数据科学家工具箱中最重要的工具。本章介绍的网格搜索与交叉验证只能应用于单个监督模型。但是我们前面看到，许多模型都需要预处理，在某些应用中（比如第 3 章人脸识别的例子），提取数据的不同表示是很有用的。下一章我们将会介绍 `Pipeline` 类，它允许我们在这些复杂的算法链上使用网格搜索与交叉验证。

注 12：我们强烈推荐阅读 Foster Provost 和 Tom Fawcett 的 *Data Science for Business* (O’Reilly) 一书来了解关于这一主题的更多信息。

第 6 章

算法链与管道

对于许多机器学习算法，你提供的特定数据表示非常重要，正如第 4 章中所述。我们在第 3 章中讲过，首先对数据进行缩放，然后手动合并特征，再利用无监督机器学习来学习特征。因此，大多数机器学习应用不仅需要应用单个算法，而且还需要将许多不同的处理步骤和机器学习模型链接在一起。本章将介绍如何使用 Pipeline 类来简化构建变换和模型链的过程。我们将重点介绍如何将 Pipeline 和 GridSearchCV 结合起来，从而同时搜索所有处理步骤中的参数。

举一个例子来说明模型链的重要性。我们知道，可以通过使用 MinMaxScaler 进行预处理来大大提高核 SVM 在 cancer 数据集上的性能。下面这些代码实现了划分数据、计算最小值和最大值、缩放数据与训练 SVM：

In[1]:

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# 加载并划分数据
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

# 计算训练数据的最小值和最大值
scaler = MinMaxScaler().fit(X_train)
```

In[2]:

```
# 对训练数据进行缩放
X_train_scaled = scaler.transform(X_train)
```

```
svm = SVC()
# 在缩放后的训练数据上学习SVM
svm.fit(X_train_scaled, y_train)
# 对测试数据进行缩放，并计算缩放后的数据的分数
X_test_scaled = scaler.transform(X_test)
print("Test score: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

Out[2]:

```
Test score: 0.95
```

6.1 用预处理进行参数选择

现在，假设我们希望利用 GridSearchCV 找到更好的 SVC 参数，正如第 5 章中所做的那样。我们应该怎么做？一种简单的方法可能如下所示：

In[3]:

```
from sklearn.model_selection import GridSearchCV
# 只是为了便于说明，不要在实践中使用这些代码！
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=5)
grid.fit(X_train_scaled, y_train)
print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
print("Best set score: {:.2f}".format(grid.score(X_test_scaled, y_test)))
print("Best parameters: ", grid.best_params_)
```

Out[3]:

```
Best cross-validation accuracy: 0.98
Best set score: 0.97
Best parameters:  {'gamma': 1, 'C': 1}
```

这里我们利用缩放后的数据对 SVC 参数进行网格搜索。但是，上面的代码中有一个不易察觉的陷阱。在缩放数据时，我们使用了**训练集中的所有数据**来找到训练的方法。然后，我们使用**缩放后的训练数据**来运行带交叉验证的网格搜索。对于交叉验证中的每次划分，原始训练集的一部分被划分为训练部分，另一部分被划分为测试部分。测试部分用于度量在训练部分上所训练的模型在新数据上的表现。但是，我们在缩放数据时已经使用过测试部分中所包含的信息。请记住，交叉验证每次划分的测试部分都是训练集的一部分，我们使用整个训练集的信息来找到数据的正确缩放。

对于模型来说，这些数据与新数据看起来截然不同。如果我们观察新数据（比如测试集中的数据），那么这些数据并没有用于对训练数据进行缩放，其最大值和最小值也可能与训练数据不同。下面这个例子（图 6-1）显示了交叉验证与最终评估这两个过程中数据处理的不同之处：

In[4]:

```
mglearn.plots.plot_improper_processing()
```

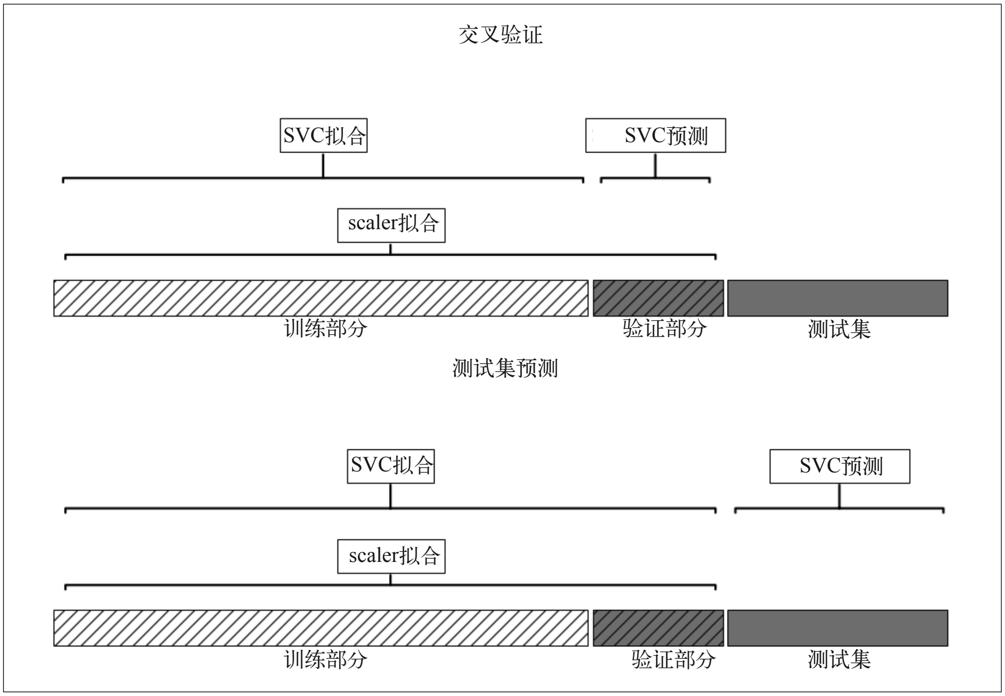


图 6-1：在交叉验证循环之外进行预处理时的数据使用情况

因此，对于建模过程，交叉验证中的划分无法正确地反映新数据的特征。我们已经将这部分数据的信息泄露（leak）给建模过程。这将导致在交叉验证过程中得到过于乐观的结果，并可能会导致选择次优的参数。

为了解决这个问题，在交叉验证的过程中，应该**在进行任何预处理之前**完成数据集的划分。任何从数据集中提取信息的处理过程都应该仅应用于数据集的训练部分，因此，任何交叉验证都应该位于处理过程的"最外层循环"。

在 `scikit-learn` 中，要想使用 `cross_val_score` 函数和 `GridSearchCV` 函数实现这一点，可以使用 `Pipeline` 类。`Pipeline` 类可以将多个处理步骤合并（glue）为单个 `scikit-learn` 估计器。`Pipeline` 类本身具有 `fit`、`predict` 和 `score` 方法，其行为与 `scikit-learn` 中的其他模型相同。`Pipeline` 类最常见的用例是将预处理步骤（比如数据缩放）与一个监督模型（比如分类器）链接在一起。

6.2　构建管道

我们来看一下如何使用 `Pipeline` 类来表示在使用 `MinMaxScaler` 缩放数据之后再训练一个 SVM 的工作流程（暂时不用网格搜索）。首先，我们构建一个由步骤列表组成的管道对象。每个步骤都是一个元组，其中包含一个名称（你选定的任意字符串[1]）和一个估计器的实例：

注 1：只有一个例外，就是该名称不能包含双下划线 __。

In[5]:

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])
```

这里我们创建了两个步骤：第一个叫作 "scaler"，是 MinMaxScaler 的实例；第二个叫作 "svm"，是 SVC 的实例。现在我们可以像任何其他 scikit-learn 估计器一样来拟合这个管道：

In[6]:

```
pipe.fit(X_train, y_train)
```

这里 pipe.fit 首先对第一个步骤（缩放器）调用 fit，然后使用该缩放器对训练数据进行变换，最后用缩放后的数据来拟合 SVM。要想在测试数据上进行评估，我们只需调用 pipe.score：

In[7]:

```
print("Test score: {:.2f}".format(pipe.score(X_test, y_test)))
```

Out[7]:

```
Test score: 0.95
```

如果对管道调用 score 方法，则首先使用缩放器对测试数据进行变换，然后利用缩放后的测试数据对 SVM 调用 score 方法。如你所见，这个结果与我们从本章开头的代码得到的结果（手动进行数据变换）是相同的。利用管道，我们减少了“预处理 + 分类”过程所需要的代码量。但是，使用管道的主要优点在于，现在我们可以在 cross_val_score 或 GridSearchCV 中使用这个估计器。

6.3 在网格搜索中使用管道

在网格搜索中使用管道的工作原理与使用任何其他估计器都相同。我们定义一个需要搜索的参数网格，并利用管道和参数网格构建一个 GridSearchCV。不过在指定参数网格时存在一处细微的变化。我们需要为每个参数指定它在管道中所属的步骤。我们要调节的两个参数 C 和 gamma 都是 SVC 的参数，属于第二个步骤。我们给这个步骤的名称是 "svm"。为管道定义参数网格的语法是为每个参数指定步骤名称，后面加上 __（双下划线），然后是参数名称。因此，要想搜索 SVC 的 C 参数，必须使用 "svm__C" 作为参数网格字典的键，对 gamma 参数也是同理：

In[8]:

```
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

有了这个参数网格，我们可以像平常一样使用 GridSearchCV：

In[9]:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
print("Test set score: {:.2f}".format(grid.score(X_test, y_test)))
print("Best parameters: {}".format(grid.best_params_))
```

Out[9]:

```
Best cross-validation accuracy: 0.98
Test set score: 0.97
Best parameters: {'svm__C': 1, 'svm__gamma': 1}
```

与前面所做的网格搜索不同，现在对于交叉验证的每次划分来说，仅使用训练部分对 MinMaxScaler 进行拟合，测试部分的信息没有泄露到参数搜索中。将图 6-2 与图 6-1 进行对比。

In[10]:

```
mglearn.plots.plot_proper_processing()
```

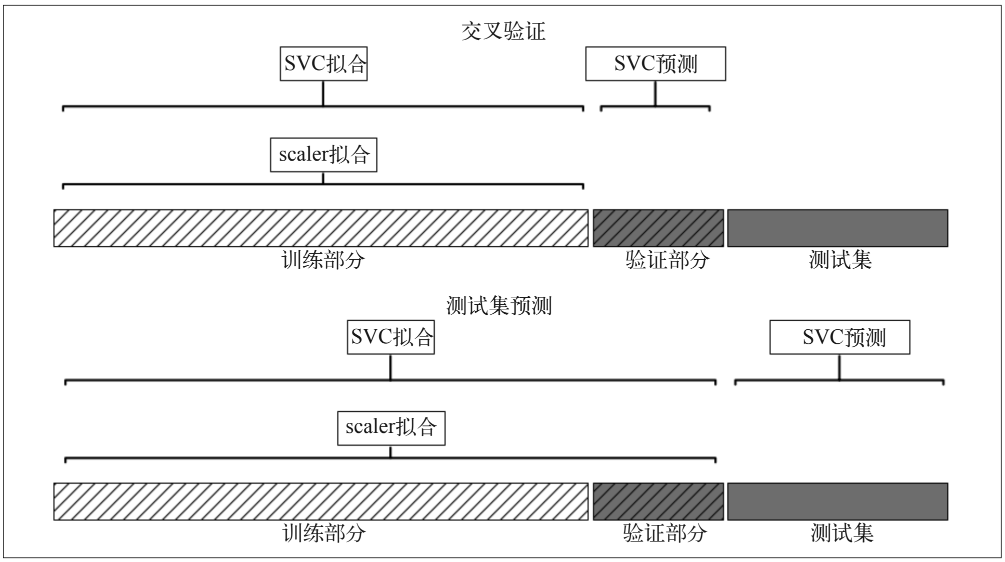


图 6-2：使用管道在交叉验证循环内部进行预处理时的数据使用情况

在交叉验证中，信息泄露的影响大小取决于预处理步骤的性质。使用测试部分来估计数据的范围，通常不会产生可怕的影响，但在特征提取和特征选择中使用测试部分，则会导致结果的显著差异。

举例说明信息泄露

在 Hastie、Tibshirani 与 Friedman 合著的《统计学习基础》一书中给出了交叉验证中信息泄露的一个很好的例子，这里我们复制了一个修改版本。我们考虑一个假想的回归任务，包含从高斯分布中独立采样的 100 个样本与 10 000 个特征。我们还从高斯分布中对响应进行采样：

In[11]:

```
rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))
```

考虑到我们创建数据集的方式，数据 X 与目标 y 之间没有任何关系（它们是独立的），所以应该不可能从这个数据集中学到任何内容。现在我们将完成下列工作。首先利用 SelectPercentile 特征选择从 10 000 个特征中选择信息量最大的特征，然后使用交叉验证对 Ridge 回归进行评估：

In[12]:

```
from sklearn.feature_selection import SelectPercentile, f_regression

select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
X_selected = select.transform(X)
print("X_selected.shape: {}".format(X_selected.shape))
```

Out[12]:

```
X_selected.shape: (100, 500)
```

In[13]:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge
print("Cross-validation accuracy (cv only on ridge): {:.2f}".format(
      np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))))
```

Out[13]:

```
Cross-validation accuracy (cv only on ridge): 0.91
```

交叉验证计算得到的平均 R^2 为 0.91，表示这是一个非常好的模型。这显然是不对的，因为我们的数据是完全随机的。这里的特征选择从 10 000 个随机特征中（碰巧）选出了与目标相关性非常好的一些特征。由于我们在交叉验证**之外**对特征选择进行拟合，所以它能够找到在训练部分和测试部分都相关的特征。从测试部分泄露出去的信息包含的信息量非常大，导致得到非常不切实际的结果。我们将这个结果与正确的交叉验证（使用管道）进行对比：

In[14]:

```
pipe = Pipeline([("select", SelectPercentile(score_func=f_regression,
                                             percentile=5)),
                 ("ridge", Ridge())])
print("Cross-validation accuracy (pipeline): {:.2f}".format(
      np.mean(cross_val_score(pipe, X, y, cv=5))))
```

Out[14]:

```
Cross-validation accuracy (pipeline): -0.25
```

这一次我们得到了**负的** R^2 分数，表示模型很差。利用管道，特征选择现在位于交叉验证循环**内部**。也就是说，仅使用数据的训练部分来选择特征，而不使用测试部分。特征选择找到的特征在训练集中与目标相关，但由于数据是完全随机的，这些特征在测试集中并不与目标相关。在这个例子中，修正特征选择中的数据泄露问题，结论也由“模型表现很好”变为“模型根本没有效果”。

6.4 通用的管道接口

Pipeline 类不但可用于预处理和分类，实际上还可以将任意数量的估计器连接在一起。例如，你可以构建一个包含特征提取、特征选择、缩放和分类的管道，总共有 4 个步骤。同样，最后一步可以用回归或聚类代替分类。

对于管道中估计器的唯一要求就是，除了最后一步之外的所有步骤都需要具有 transform 方法，这样它们可以生成新的数据表示，以供下一个步骤使用。

在调用 Pipeline.fit 的过程中，管道内部依次对每个步骤调用 fit 和 transform[2]，其输入是前一个步骤中 transform 方法的输出。对于管道中的最后一步，则仅调用 fit。

忽略某些细枝末节，其实现方法如下所示。请记住，pipeline.steps 是由元组组成的列表，所以 pipeline.steps[0][1] 是第一个估计器，pipeline.steps[1][1] 是第二个估计器，以此类推：

In[15]:

```
def fit(self, X, y):
    X_transformed = X
    for name, estimator in self.steps[:-1]:
        # 遍历除最后一步之外的所有步骤
        # 对数据进行拟合和变换
        X_transformed = estimator.fit_transform(X_transformed, y)
    # 对最后一步进行拟合
    self.steps[-1][1].fit(X_transformed, y)
    return self
```

使用 Pipeline 进行预测时，我们同样利用除最后一步之外的所有步骤对数据进行变换（transform），然后对最后一步调用 predict：

In[16]:

```
def predict(self, X):
    X_transformed = X
    for step in self.steps[:-1]:
        # 遍历除最后一步之外的所有步骤
        # 对数据进行变换
        X_transformed = step[1].transform(X_transformed)
# 利用最后一步进行预测
return self.steps[-1][1].predict(X_transformed)
```

整个过程如图 6-3 所示，其中包含两个变换器（transformer）T1 和 T2，还有一个分类器（叫作 Classifier）。

注 2：或仅调用 fit_transform。

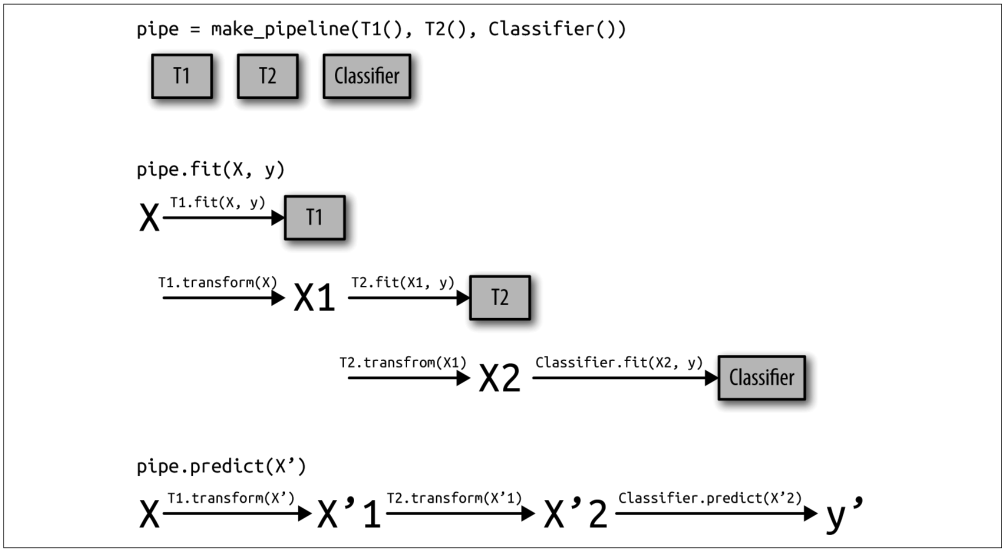


图 6-3：管道的训练和预测过程概述

管道实际上比上图更加通用。管道的最后一步不需要具有 predict 函数，比如说，我们可以创建一个只包含一个缩放器和一个 PCA 的管道。由于最后一步（PCA）具有 transform 方法，所以我们可以对管道调用 transform，以得到将 PCA.transform 应用于前一个步骤处理过的数据后得到的输出。管道的最后一步只需要具有 fit 方法。

6.4.1 用make_pipeline方便地创建管道

利用上述语法创建管道有时有点麻烦，我们通常不需要为每一个步骤提供用户指定的名称。有一个很方便的函数 make_pipeline，可以为我们创建管道并根据每个步骤所属的类为其自动命名。make_pipeline 的语法如下所示：

In[17]:

```
from sklearn.pipeline import make_pipeline
# 标准语法
pipe_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])
# 缩写语法
pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
```

管道对象 pipe_long 和 pipe_short 的作用完全相同，但 pipe_short 的步骤是自动命名的。我们可以通过查看 steps 属性来查看步骤的名称：

In[18]:

```
print("Pipeline steps:\n{}".format(pipe_short.steps))
```

Out[18]:

```
Pipeline steps:
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))),
 ('svc', SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
```

```
                 decision_function_shape=None, degree=3, gamma='auto',
                 kernel='rbf', max_iter=-1, probability=False,
                 random_state=None, shrinking=True, tol=0.001,
                 verbose=False))]
```

这两个步骤被命名为 minmaxscaler 和 svc。一般来说，步骤名称只是类名称的小写版本。如果多个步骤属于同一个类，则会附加一个数字：

In[19]:

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())
print("Pipeline steps:\n{}".format(pipe.steps))
```

Out[19]:

```
Pipeline steps:
[('standardscaler-1', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('pca', PCA(copy=True, iterated_power=4, n_components=2, random_state=None,
             svd_solver='auto', tol=0.0, whiten=False)),
 ('standardscaler-2', StandardScaler(copy=True, with_mean=True, with_std=True))]
```

如你所见，第一个 StandardScaler 步骤被命名为 standardscaler-1，而第二个被命名为 standardscaler-2。但在这种情况下，使用具有明确名称的 Pipeline 构建可能更好，以便为每个步骤提供更具语义的名称。

6.4.2　访问步骤属性

通常来说，你希望检查管道中某一步骤的属性——比如线性模型的系数或 PCA 提取的成分。要想访问管道中的步骤，最简单的方法是通过 named_steps 属性，它是一个字典，将步骤名称映射为估计器：

In[20]:

```
# 用前面定义的管道对cancer数据集进行拟合
pipe.fit(cancer.data)
# 从"pca"步骤中提取前两个主成分
components = pipe.named_steps["pca"].components_
print("components.shape: {}".format(components.shape))
```

Out[20]:

```
components.shape: (2, 30)
```

6.4.3　访问网格搜索管道中的属性

本章前面说过，使用管道的主要原因之一就是进行网格搜索。一个常见的任务是在网格搜索内访问管道的某些步骤。我们对 cancer 数据集上的 LogisticRegression 分类器进行网格搜索，在将数据传入 LogisticRegression 分类器之前，先用 Pipeline 和 StandardScaler 对数据进行缩放。首先，我们用 make_pipeline 函数创建一个管道：

In[21]:

```
from sklearn.linear_model import LogisticRegression

pipe = make_pipeline(StandardScaler(), LogisticRegression())
```

接下来，我们创建一个参数网格。我们在第 2 章中说过，LogisticRegression 需要调节的正则化参数是参数 C。我们对这个参数使用对数网格，在 0.01 和 100 之间进行搜索。由于我们使用了 make_pipeline 函数，所以管道中 LogisticRegression 步骤的名称是小写的类名称 logisticregression。因此，为了调节参数 C，我们必须指定 logisticregression__C 的参数网格：

In[22]:

```
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}
```

像往常一样，我们将 cancer 数据集划分为训练集和测试集，并对网格搜索进行拟合：

In[23]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=4)
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

那么我们如何访问 GridSearchCV 找到的最佳 LogisticRegression 模型的系数呢？我们从第 5 章中知道，GridSearchCV 找到的最佳模型（在所有训练数据上训练得到的模型）保存在 grid.best_estimator_ 中：

In[24]:

```
print("Best estimator:\n{}".format(grid.best_estimator_))
```

Out[24]:

```
Best estimator:
Pipeline(steps=[
    ('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True)),
    ('logisticregression', LogisticRegression(C=0.1, class_weight=None,
    dual=False, fit_intercept=True, intercept_scaling=1, max_iter=100,
    multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
    solver='liblinear', tol=0.0001, verbose=0, warm_start=False))])
```

在我们的例子中，best_estimator_ 是一个管道，它包含两个步骤：standardscaler 和 logisticregression。如前所述，我们可以使用管道的 named_steps 属性来访问 logisticregression 步骤：

In[25]:

```
print("Logistic regression step:\n{}".format(
      grid.best_estimator_.named_steps["logisticregression"]))
```

Out[25]:

```
Logistic regression step:
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
                  intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                  penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                  verbose=0, warm_start=False)
```

现在我们得到了训练过的 LogisticRegression 实例，下面我们可以访问与每个输入特征相关的系数（权重）：

In[26]:

```
print("Logistic regression coefficients:\n{}".format(
      grid.best_estimator_.named_steps["logisticregression"].coef_))
```

Out[26]:

```
Logistic regression coefficients:
[[-0.389 -0.375 -0.376 -0.396 -0.115  0.017 -0.355 -0.39  -0.058  0.209
  -0.495 -0.004 -0.371 -0.383 -0.045  0.198  0.004 -0.049  0.21   0.224
  -0.547 -0.525 -0.499 -0.515 -0.393 -0.123 -0.388 -0.417 -0.325 -0.139]]
```

这个系数列表可能有点长，但它通常有助于理解你的模型。

6.5　网格搜索预处理步骤与模型参数

我们可以利用管道将机器学习工作流程中的所有处理步骤封装成一个 scikit-learn 估计器。这么做的另一个好处在于，现在我们可以使用监督任务（比如回归或分类）的输出来**调节预处理参数**。在前几章里，我们在应用岭回归之前使用了 boston 数据集的多项式特征。下面我们用一个管道来重复这个建模过程。管道包含 3 个步骤：缩放数据、计算多项式特征与岭回归：

In[27]:

```
from sklearn.datasets import load_boston
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target,
                                                    random_state=0)

from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(),
    Ridge())
```

我们怎么知道选择几次多项式，或者是否选择多项式或交互项呢？理想情况下，我们希望根据分类结果来选择 degree 参数。我们可以利用管道搜索 degree 参数以及 Ridge 的 alpha 参数。为了做到这一点，我们要定义一个包含这两个参数的 param_grid，并用步骤名称作为前缀：

In[28]:

```
param_grid = {'polynomialfeatures__degree': [1, 2, 3],
              'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

现在我们可以再次运行网格搜索：

In[29]:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)
```

像第 5 章中所做的那样，我们可以用热图将交叉验证的结果可视化（见图 6-4）：

In[30]:

```
plt.matshow(grid.cv_results_['mean_test_score'].reshape(3, -1),
            vmin=0, cmap="viridis")
plt.xlabel("ridge__alpha")
plt.ylabel("polynomialfeatures__degree")
plt.xticks(range(len(param_grid['ridge__alpha'])), param_grid['ridge__alpha'])
plt.yticks(range(len(param_grid['polynomialfeatures__degree'])),
           param_grid['polynomialfeatures__degree'])

plt.colorbar()
```

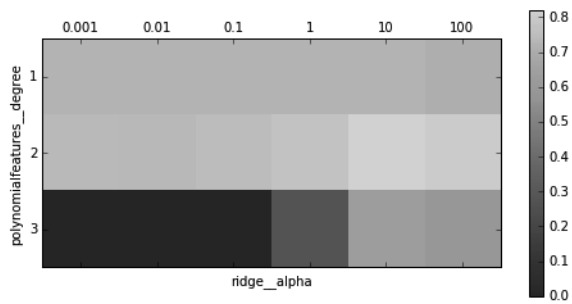


图 6-4：以多项式特征的次数和岭回归的 alpha 参数为坐标轴，绘制交叉验证平均分数的热图

从交叉验证的结果中可以看出，使用二次多项式很有用，但三次多项式的效果比一次或二次都要差很多。从找到的最佳参数中也可以看出这一点：

In[31]:

```
print("Best parameters: {}".format(grid.best_params_))
```

Out[31]:

```
Best parameters: {'polynomialfeatures__degree': 2, 'ridge__alpha': 10}
```

这个最佳参数对应的分数如下：

In[32]:

```
print("Test-set score: {:.2f}".format(grid.score(X_test, y_test)))
```

Out[32]:

```
Test-set score: 0.77
```

为了对比，我们运行一个没有多项式特征的网格搜索：

In[33]:

```
param_grid = {'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = make_pipeline(StandardScaler(), Ridge())
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
print("Score without poly features: {:.2f}".format(grid.score(X_test, y_test)))
```

Out[33]:

```
Score without poly features: 0.63
```

正与我们观察图 6-4 中的网格搜索结果所预料的那样，不使用多项式特征得到了明显更差的结果。

同时搜索预处理参数与模型参数是一个非常强大的策略。但是要记住，GridSearchCV 会尝试指定参数的**所有可能组合**。因此，向网格中添加更多参数，需要构建的模型数量将呈指数增长。

6.6 网格搜索选择使用哪个模型

你甚至可以进一步将 GridSearchCV 和 Pipeline 结合起来：还可以搜索管道中正在执行的实际步骤（比如用 StandardScaler 还是用 MinMaxScaler）。这样会导致更大的搜索空间，应该予以仔细考虑。尝试所有可能的解决方案，通常并不是一种可行的机器学习策略。但下面是一个例子：在 iris 数据集上比较 RandomForestClassifier 和 SVC。我们知道，SVC 可能需要对数据进行缩放，所以我们还需要搜索是使用 StandardScaler 还是不使用预处理。我们知道，RandomForestClassifier 不需要预处理。我们先定义管道。这里我们显式地对步骤命名。我们需要两个步骤，一个用于预处理，然后是一个分类器。我们可以用 SVC 和 StandardScaler 来将其实例化：

In[34]:

```
pipe = Pipeline([('preprocessing', StandardScaler()), ('classifier', SVC())])
```

现在我们可以定义需要搜索的 parameter_grid。我们希望 classifier 是 RandomForestClassifier 或 SVC。由于这两种分类器需要调节不同的参数，并且需要不同的预处理，所以我们可以使用 5.2.3 节“在非网格的空间中搜索”中所讲的搜索网格列表。为了将一个估计器分配给一个步骤，我们使用步骤名称作为参数名称。如果我们想跳过管道中的某个步骤（例如，RandomForest 不需要预处理），则可以将该步骤设置为 None：

In[35]:

```
from sklearn.ensemble import RandomForestClassifier

param_grid = [
    {'classifier': [SVC()], 'preprocessing': [StandardScaler(), None],
     'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
     'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100]},
    {'classifier': [RandomForestClassifier(n_estimators=100)],
     'preprocessing': [None], 'classifier__max_features': [1, 2, 3]}]
```

现在，我们可以像前面一样将网格搜索实例化并在 cancer 数据集上运行：

In[36]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

```
print("Best params:\n{}\n".format(grid.best_params_))
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Test-set score: {:.2f}".format(grid.score(X_test, y_test)))
```

Out[36]:

```
Best params:
{'classifier':
 SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
     decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',
     max_iter=-1, probability=False, random_state=None, shrinking=True,
     tol=0.001, verbose=False),
 'preprocessing':
 StandardScaler(copy=True, with_mean=True, with_std=True),
 'classifier__C': 10, 'classifier__gamma': 0.01}

Best cross-validation score: 0.99
Test-set score: 0.98
```

网格搜索的结果是 SVC 与 StandardScaler 预处理，在 C=10 和 gamma=0.01 时给出最佳结果。

6.7　小结与展望

本章介绍了 Pipeline 类，这是一种通用工具，可以将机器学习工作流程中的多个处理步骤链接在一起。现实世界中的机器学习应用很少仅涉及模型的单独使用，而是需要一系列处理步骤。使用管道可以将多个步骤封装为单个 Python 对象，这个对象具有我们熟悉的 scikit-learn 接口 fit、predict 和 transform。特别是使用交叉验证进行模型评估与使用网格搜索进行参数选择时，使用 Pipeline 类来包括所有处理步骤对正确的评估至关重要。利用 Pipeline 类还可以让代码更加简洁，并减少不用 pipeline 类构建处理链时可能会犯的错误（比如忘记将所有变换器应用于测试集，或者应用顺序错误）的可能性。选择特征提取、预处理和模型的正确组合，这在某种程度上是一门艺术，通常需要一些试错。但是有了管道，这种“尝试”多个不同的处理步骤是非常简单的。在进行试验时，要小心不要将处理过程复杂化，并且一定要评估一下模型中的每个组件是否必要。

学完本章，我们已经学完了 scikit-learn 提供的所有通用工具与算法。现在你已经掌握了所有必要的技术，并了解了在实践中应用机器学习的必要机制。下一章我们将深入一种在实践中常见的数据类型——文本数据，它需要具备一些专门的知识才能正确处理。

第 7 章

处理文本数据

在第 4 章中，我们讨论过表示数据属性的两种类型的特征：连续特征与分类特征，前者用于描述数量，后者是固定列表中的元素。在许多应用中还可以见到第三种类型的特征：文本。举个例子，如果我们想要判断一封电子邮件是合法邮件还是垃圾邮件，那么邮件内容一定会包含对这个分类任务非常重要的信息。或者，我们可能想要了解一位政治家对移民问题的看法。这个人的演讲或推文可能会提供有用的信息。在客户服务中，我们通常想知道一条消息是投诉还是咨询。我们可以利用消息的主题和内容来自动判断客户的目的，从而将消息发送给相关部门，甚至可以发送一封全自动回复。

文本数据通常被表示为由字符组成的字符串。在上面给出的所有例子中，文本数据的长度都不相同。这个特征显然与前面讨论过的数值特征有很大不同，我们需要先处理数据，然后才能对其应用机器学习算法。

7.1　用字符串表示的数据类型

在深入研究表示机器学习文本数据的处理步骤之前，我们希望简要讨论你可能会遇到的不同类型的文本数据。文本通常只是数据集中的字符串，但并非所有的字符串特征都应该被当作文本来处理。我们在第 5 章讨论过，字符串特征有时可以表示分类变量。在查看数据之前，我们无法知道如何处理一个字符串特征。

你可能会遇到四种类型的字符串数据：

- 分类数据
- 可以在语义上映射为类别的自由字符串
- 结构化字符串数据
- 文本数据

分类数据（categorical data）是来自固定列表的数据。比如你通过调查人们最喜欢的颜色来收集数据，你向他们提供了一个下拉菜单，可以从“红色”“绿色”“蓝色”“黄色”“黑色”“白色”“紫色”和“粉色”中选择。这样会得到一个包含 8 个不同取值的数据集，这 8 个不同取值表示的显然是分类变量。你可以通过观察来判断你的数据是不是分类数据（如果你看到了许多不同的字符串，那么不太可能是分类变量），并通过计算数据集中的唯一值并绘制其出现次数的直方图来验证你的判断。你可能还希望检查每个变量是否实际对应于一个在应用中有意义的分类。调查过程进行到一半，有人可能发现调查问卷中将“black”（黑色）错拼为“blak”，并随后对其进行了修改。因此，你的数据集中同时包含“black”和“blak”，它们对应于相同的语义，所以应该将二者合并。

现在想象一下，你向用户提供的不是一个下拉菜单，而是一个文本框，让他们填写自己最喜欢的颜色。许多人的回答可能是像“黑色”或“蓝色”之类的颜色名称。其他人可能会出现笔误，使用不同的单词拼写（比如“gray”和“grey”[1]），或使用更加形象的具体名称（比如“午夜蓝色”）。你还会得到一些非常奇怪的条目。xkcd 颜色调查（https://blog.xkcd.com/2010/05/03/color-survey-results/）中有一些很好的例子，其中有人为颜色命名，给出了如“迅猛龙泄殖腔”和“我牙医办公室的橙色。我仍然记得他的头皮屑慢慢地漂落到我张开的下巴”之类的名称，很难将这些名称与颜色自动对应（或者根本就无法对应）。从文本框中得到的回答属于上述列表中的第二类，**可以在语义上映射为类别的自由字符串**（free strings that can be semantically mapped to categories）。可能最好将这种数据编码为分类变量，你可以利用最常见的条目来选择类别，也可以自定义类别，使用户回答对应用有意义。这样你可能会有一些标准颜色的类别，可能还有一个“多色”类别（对于像“绿色与红色条纹”之类的回答）和“其他”类别（对于无法归类的回答）。这种字符串预处理过程可能需要大量的人力，并且不容易自动化。如果你能够改变数据的收集方式，那么我们强烈建议，对于分类变量能够更好表示的概念，不要使用手动输入值。

通常来说，手动输入值不与固定的类别对应，但仍有一些内在的**结构**（structure），比如地址、人名或地名、日期、电话号码或其他标识符。这种类型的字符串通常难以解析，其处理方法也强烈依赖于上下文和具体领域。对这种情况的系统处理方法超出了本书的范围。

最后一类字符串数据是自由格式的**文本数据**（text data），由短语或句子组成。例子包括推文、聊天记录和酒店评论，还包括莎士比亚文集、维基百科的内容或古腾堡计划收集的 50 000 本电子书。所有这些集合包含的信息大多是由单词组成的句子。[2] 为了简单起见，我们假设所有的文档都只使用一种语言：英语。[3] 在文本分析的语境中，数据集通常被称为**语料库**（corpus），每个由单个文本表示的数据点被称为**文档**（document）。这些术语来自于**信息检索**（information retrieval，IR）和**自然语言处理**（natural language processing，NLP）的社区，它们主要针对文本数据。

注 1：两个词的意思都是“灰色”。——译者注

注 2：推文中链接网站的内容所包含的信息可能比推文本身还要多。

注 3：本章接下来的绝大部分内容也适用于其他使用罗马字母的语言，部分适用于具有单词定界符的其他语言。例如，中文没有词边界，还有其他方面的挑战，所以难以应用本章介绍的技术。

7.2 示例应用：电影评论的情感分析

作为本章的一个运行示例，我们将使用由斯坦福研究员 Andrew Maas 收集的 IMDb（Internet Movie Database，互联网电影数据库）网站的电影评论数据集。[4] 这个数据集包含评论文本，还有一个标签，用于表示该评论是“正面的”（positive）还是“负面的”（negative）。IMDb 网站本身包含从 1 到 10 的打分。为了简化建模，这些评论打分被归纳为一个二分类数据集，评分大于等于 7 的评论被标记为“正面的”，评分小于等于 4 的评论被标记为“负面的”，中性评论没有包含在数据集中。我们不讨论这种方法是否是一种好的数据表示，而只是使用 Andrew Maas 提供的数据。

将数据解压之后，数据集包括两个独立文件夹中的文本文件，一个是训练数据，一个是测试数据。每个文件夹又都有两个子文件夹，一个叫作 pos，一个叫作 neg：[5]

In[2]:

```
!tree -L 2 data/aclImdb
```

Out[2]:

```
data/aclImdb
├── test
│   ├── neg
│   └── pos
└── train
    ├── neg
    └── pos
6 directories, 0 files
```

pos 文件夹包含所有正面的评论，每条评论都是一个单独的文本文件，neg 文件夹与之类似。`scikit-learn` 中有一个辅助函数可以加载用这种文件夹结构保存的文件，其中每个子文件夹对应于一个标签，这个函数叫作 `load_files`。我们首先将 `load_files` 函数应用于训练数据：

In[3]:

```
from sklearn.datasets import load_files

reviews_train = load_files("data/aclImdb/train/")
# load_files返回一个Bunch对象，其中包含训练文本和训练标签
text_train, y_train = reviews_train.data, reviews_train.target
print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))
print("text_train[1]:\n{}".format(text_train[1]))
```

Out[3]:

```
type of text_train:  <class 'list'>
length of text_train:  25000
text_train[1]:
b'Words can\'t describe how bad this movie is. I can\'t explain it by writing
```

注 4：你可以在 http://ai.stanford.edu/~amaas/data/sentiment/ 下载这个数据集。

注 5：v1.0 版数据的目录结构与这个略有不同。——译者注

```
only. You have too see it for yourself to get at grip of how horrible a movie
really can be. Not that I recommend you to do that. There are so many
clich\xc3\xa9s, mistakes (and all other negative things you can imagine) here
that will just make you cry. To start with the technical first, there are a
LOT of mistakes regarding the airplane. I won\'t list them here, but just
mention the coloring of the plane. They didn\'t even manage to show an
airliner in the colors of a fictional airline, but instead used a 747
painted in the original Boeing livery. Very bad. The plot is stupid and has
been done many times before, only much, much better. There are so many
ridiculous moments here that i lost count of it really early. Also, I was on
the bad guys\' side all the time in the movie, because the good guys were so
stupid. "Executive Decision" should without a doubt be you\'re choice over
this one, even the "Turbulence"-movies are better. In fact, every other
movie in the world is better than this one.'
```

你可以看到，text_train 是一个长度为 25 000 的列表[6]，其中每个元素是包含一条评论的字符串。我们打印出索引编号为 1 的评论。你还可以看到，评论中包含一些 HTML 换行符（
）。虽然这些符号不太可能对机器学习模型产生很大影响，但最好在继续下一步之前清洗数据并删除这种格式：

In[4]:

```
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

text_train 的元素类型与你所使用的 Python 版本有关。在 Python 3 中，它们是 bytes 类型，是表示字符串数据的二进制编码。在 Python 2 中，text_train 包含的是字符串。这里不会深入讲解 Python 中不同的字符串类型，但我们推荐阅读 Python 2（https://docs.python.org/2/howto/unicode.html）和 Python 3（https://docs.python.org/3/howto/unicode.html）的文档中关于字符串和 Unicode 的内容。

收集数据集时保持正类和反类的平衡，这样所有正面字符串和负面字符串的数量相等：

In[5]:

```
print("Samples per class (training): {}".format(np.bincount(y_train)))
```

Out[5]:

```
Samples per class (training): [12500 12500]
```

我们用同样的方式加载测试数据集：

In[6]:

```
reviews_test = load_files("data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Number of documents in test data: {}".format(len(text_test)))
print("Samples per class (test): {}".format(np.bincount(y_test)))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

Out[6]:

```
Number of documents in test data: 25000
Samples per class (test): [12500 12500]
```

注 6：对于 v1.0 版数据，其训练集大小是 75 000，而不是 25 000，因为其中还包含 50 000 个用于无监督学习的无标签文档。在进行后续操作之前，建议先将这 50 000 个无标签文档从训练集中剔除。——译者注

我们要解决的任务如下：给定一条评论，我们希望根据该评论的文本内容对其分配一个“正面的”或“负面的”标签。这是一项标准的二分类任务。但是，文本数据并不是机器学习模型可以处理的格式。我们需要将文本的字符串表示转换为数值表示，从而可以对其应用机器学习算法。

7.3 将文本数据表示为词袋

用于机器学习的文本表示有一种最简单的方法，也是最有效且最常用的方法，就是使用**词袋**（bag-of-words）表示。使用这种表示方式时，我们舍弃了输入文本中的大部分结构，如章节、段落、句子和格式，只计算语料库中**每个单词在每个文本中的出现频次**。舍弃结构并仅计算单词出现次数，这会让脑海中出现将文本表示为“袋”的画面。

对于文档语料库，计算词袋表示包括以下三个步骤。

(1) **分词**（tokenization）。将每个文档划分为出现在其中的单词 [称为词例（token）]，比如按空格和标点划分。
(2) 构建词表（vocabulary building）。收集一个词表，里面包含出现在任意文档中的所有词，并对它们进行编号（比如按字母顺序排序）。
(3) **编码**（encoding）。对于每个文档，计算词表中每个单词在该文档中的出现频次。

在步骤 1 和步骤 2 中涉及一些细微之处，我们将在本章后面进一步深入讨论。目前，我们来看一下如何利用 `scikit-learn` 来应用词袋处理过程。图 7-1 展示了对字符串 `"This is how you get ants."` 的处理过程。其输出是包含每个文档中单词计数的一个向量。对于词表中的每个单词，我们都有它在每个文档中的出现次数。也就是说，整个数据集中的每个唯一单词都对应于这种数值表示的一个特征。请注意，原始字符串中的单词顺序与词袋特征表示完全无关。

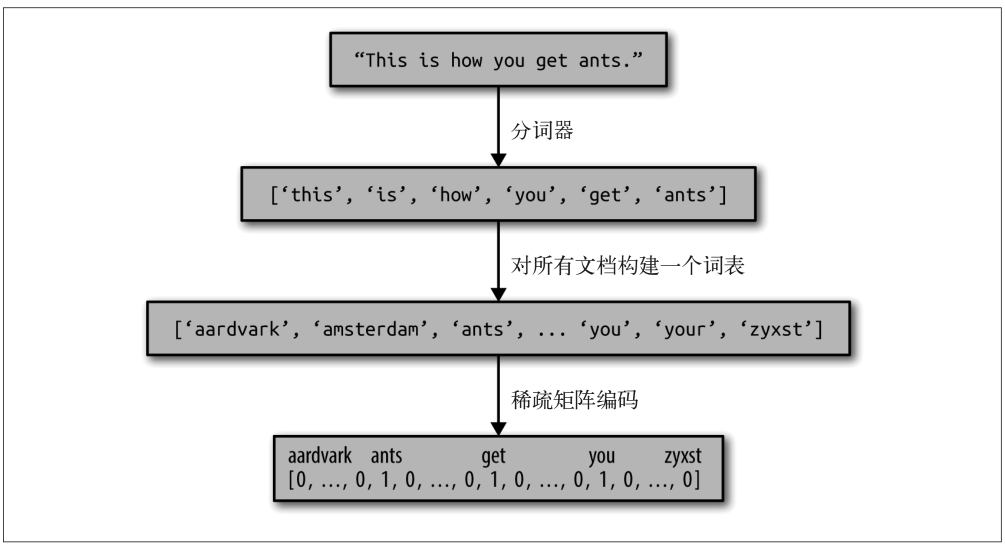


图 7-1：词袋处理过程

7.3.1 将词袋应用于玩具数据集

词袋表示是在 CountVectorizer 中实现的，它是一个变换器（transformer）。我们首先将它应用于一个包含两个样本的玩具数据集，来看一下它的工作原理：

In[7]:

```
bards_words =["The fool doth think he is wise,",
              "but the wise man knows himself to be a fool"]
```

我们导入 CountVectorizer 并将其实例化，然后对玩具数据进行拟合，如下所示：

In[8]:

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

拟合 CountVectorizer 包括训练数据的分词与词表的构建，我们可以通过 vocabulary_ 属性来访问词表：

In[9]:

```
print("Vocabulary size: {}".format(len(vect.vocabulary_)))
print("Vocabulary content:\n {}".format(vect.vocabulary_))
```

Out[9]:

```
Vocabulary size: 13
Vocabulary content:
 {'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2, 'to': 11, 'knows': 7,
  'man': 8, 'fool': 3, 'is': 6, 'be': 0, 'think': 10, 'but': 1}
```

词表共包含 13 个词，从 "be" 到 "wise"。

我们可以调用 transform 方法来创建训练数据的词袋表示：

In[10]:

```
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))
```

Out[10]:

```
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'
    with 16 stored elements in Compressed Sparse Row format>
```

词袋表示保存在一个 SciPy 稀疏矩阵中，这种数据格式只保存非零元素（参见第 1 章）。这个矩阵的形状为 2×13，每行对应于两个数据点之一，每个特征对应于词表中的一个单词。这里使用稀疏矩阵，是因为大多数文档都只包含词表中的一小部分单词，也就是说，特征数组中的大部分元素都为 0。想想看，与所有英语单词（这是词表的建模对象）相比，一篇电影评论中可能出现多少个不同的单词。保存所有 0 的代价很高，也浪费内存。要想查看稀疏矩阵的实际内容，可以使用 toarray 方法将其转换为“密集的”NumPy 数组（保存所有 0 元素）：[7]

注 7：这么做之所以是可行的，是因为我们使用的是仅包含 13 个单词的小型玩具数据集。对于任何真实数据集来说，这将会导致 MemoryError（内存错误）。

In[11]:

```
print("Dense representation of bag_of_words:\n{}".format(
    bag_of_words.toarray()))
```

Out[11]:

```
Dense representation of bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

我们可以看到，每个单词的计数都是 0 或 1。bards_words 中的两个字符串都没有包含相同的单词。我们来看一下如何阅读这些特征向量。第一个字符串（"The fool doth think he is wise,"）被表示为第一行，对于词表中的第一个单词 "be"，出现 0 次。对于词表中的第二个单词 "but"，出现 0 次。对于词表中的第三个单词 "doth"，出现 1 次，以此类推。通过观察这两行可以看出，第 4 个单词 "fool"、第 10 个单词 "the" 与第 13 个单词 "wise" 同时出现在两个字符串中。

7.3.2　将词袋应用于电影评论

上一节我们详细介绍了词袋处理过程，下面我们将其应用于电影评论情感分析的任务。前面我们将 IMDb 评论的训练数据和测试数据加载为字符串列表（text_train 和 text_test），现在我们将处理它们：

In[12]:

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))
```

Out[12]:

```
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'
    with 3431196 stored elements in Compressed Sparse Row format>
```

X_train 是训练数据的词袋表示，其形状为 25 000×74 849，这表示词表中包含 74 849 个元素。数据同样被保存为 SciPy 稀疏矩阵。我们来更详细地看一下这个词表。访问词表的另一种方法是使用向量器（vectorizer）的 get_feature_name 方法，它将返回一个列表，每个元素对应于一个特征：

In[13]:

```
feature_names = vect.get_feature_names()
print("Number of features: {}".format(len(feature_names)))
print("First 20 features:\n{}".format(feature_names[:20]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 2000th feature:\n{}".format(feature_names[::2000]))
```

Out[13]:

```
Number of features: 74849
First 20 features:
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830',
 '006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s',
```

```
 '01', '01pm', '02']
Features 20010 to 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback',
 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
 'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
Every 2000th feature:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'beête', 'chicanery',
 'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freezer',
 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz', 'lawful',
 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher',
 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
 'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

如你所见，词表的前 10 个元素都是数字，这可能有些出人意料。所有这些数字都出现在评论中的某处，因此被提取为单词。大部分数字都没有一目了然的语义，除了 "007"，在电影的特定语境中它可能指的是詹姆斯·邦德（James Bond）这个角色。[8] 从无意义的“单词”中挑出有意义的有时很困难。进一步观察这个词表，我们发现许多以“dra”开头的英语单词。你可能注意到了，对于 "draught"、"drawback" 和 "drawer"，其单数和复数形式都包含在词表中，并且作为不同的单词。这些单词具有密切相关的语义，将它们作为不同的单词进行计数（对应于不同的特征）可能不太合适。

在尝试改进特征提取之前，我们先通过实际构建一个分类器来得到性能的量化度量。我们将训练标签保存在 y_train 中，训练数据的词袋表示保存在 X_train 中，因此我们可以在这个数据上训练一个分类器。对于这样的高维稀疏数据，类似 LogisticRegression 的线性模型通常效果最好。

我们首先使用交叉验证对 LogisticRegression 进行评估：[9]

In[14]:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("Mean cross-validation accuracy: {:.2f}".format(np.mean(scores)))
```

Out[14]:

```
Mean cross-validation accuracy: 0.88
```

我们得到的交叉验证平均分数是 88%，这对于平衡的二分类任务来说是一个合理的性能。我们知道，LogisticRegression 有一个正则化参数 C，我们可以通过交叉验证来调节它：

In[15]:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
```

注 8：对数据的快速分析可以证实这一点。你可以自己尝试验证一下。

注 9：细心的读者可能会注意到，我们这里违背了第 6 章中关于交叉验证与预处理的内容。CountVectorizer 的默认设置实际上不会收集任何统计信息，所以我们的结果是有效的。对于应用而言，从一开始就使用 Pipeline 是更好的选择，但我们后面再这么做，这里是为了便于说明。

```
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters: ", grid.best_params_)
```

Out[15]:

```
Best cross-validation score: 0.89
Best parameters:  {'C': 0.1}
```

我们使用 C=0.1 得到的交叉验证分数是 89%。现在，我们可以在测试集上评估这个参数设置的泛化性能：

In[16]:

```
X_test = vect.transform(text_test)
print("{:.2f}".format(grid.score(X_test, y_test)))
```

Out[16]:

```
0.88
```

下面我们来看一下能否改进单词提取。CountVectorizer 使用正则表达式提取词例。默认使用的正则表达式是 "\b\w\w+\b"。如果你不熟悉正则表达式，它的含义是找到所有包含至少两个字母或数字（\w）且被词边界（\b）分隔的字符序列。它不会匹配只有一个字母的单词，还会将类似"doesn't"或"bit.ly"之类的缩写分开，但它会将"h8ter"匹配为一个单词。然后，CountVectorizer 将所有单词转换为小写字母，这样"soon""Soon"和"sOon"都对应于同一个词例（因此也对应于同一个特征）。这一简单机制在实践中的效果很好，但正如前面所见，我们得到了许多不包含信息量的特征（比如数字）。减少这种特征的一种方法是，仅使用至少在 2 个文档（或者至少 5 个，等等）中出现过的词例。仅在一个文档中出现的词例不太可能出现在测试集中，因此没什么用。我们可以用 min_df 参数来设置词例至少需要在多少个文档中出现过：

In[17]:

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train with min_df: {}".format(repr(X_train)))
```

Out[17]:

```
X_train with min_df: <25000x27271 sparse matrix of type '<class 'numpy.int64'>'
    with 3354014 stored elements in Compressed Sparse Row format>
```

通过要求每个词例至少在 5 个文档中出现过，我们可以将特征数量减少到 27 271 个，正如上面的输出所示——只有原始特征的三分之一左右。我们再来查看一些词例：

In[18]:

```
feature_names = vect.get_feature_names()

print("First 50 features:\n{}".format(feature_names[:50]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 700th feature:\n{}".format(feature_names[::700]))
```

Out[18]:

```
First 50 features:
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08',
```

```
 '09', '10', '100', '1000', '100th', '101', '102', '103', '104', '105', '107',
 '108', '10s', '10th', '11', '110', '112', '116', '117', '11th', '12', '120',
 '12th', '13', '135', '13th', '14', '140', '14th', '15', '150', '15th', '16',
 '160', '1600', '16mm', '16s', '16th']
Features 20010 to 20030:
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions',
 'repetitious', 'repetitive', 'rephrase', 'replace', 'replaced', 'replacement',
 'replaces', 'replacing', 'replay', 'replayable', 'replayed', 'replaying',
 'replays', 'replete', 'replica']
Every 700th feature:
['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered',
 'cheese', 'commitment', 'courts', 'deconstructed', 'disgraceful', 'dvds',
 'eschews', 'fell', 'freezer', 'goriest', 'hauser', 'hungary', 'insinuate',
 'juggle', 'leering', 'maelstrom', 'messiah', 'music', 'occasional', 'parking',
 'pleasantville', 'pronunciation', 'recipient', 'reviews', 'sas', 'shea',
 'sneers', 'steiger', 'swastika', 'thrusting', 'tvs', 'vampyre', 'westerns']
```

数字的个数明显变少了，有些生僻词或拼写错误似乎也都消失了。我们再次运行网格搜索来看一下模型的性能如何：

In[19]:

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

Out[19]:

```
Best cross-validation score: 0.89
```

网格搜索的最佳验证精度还是 89%，这和前面一样。我们并没有改进模型，但减少要处理的特征数量可以加速处理过程，舍弃无用的特征也可能提高模型的可解释性。

> 如果一个文档中包含训练数据中没有包含的单词，并对其调用 `CountVectorizer` 的 `transform` 方法，那么这些单词将被忽略，因为它们没有包含在字典中。这对分类来说不是一个问题，因为从不在训练数据中的单词中学不到任何内容。但对于某些应用而言（比如垃圾邮件检测），添加一个特征来表示特定文档中有多少个所谓“词表外”单词可能会有所帮助。为了实现这一点，你需要设置 `min_df`，否则这个特征在训练期间永远不会被用到。

7.4 停用词

删除没有信息量的单词还有另一种方法，就是舍弃那些出现次数太多以至于没有信息量的单词。有两种主要方法：使用特定语言的停用词（stopword）列表，或者舍弃那些出现过于频繁的单词。`scikit-learn` 的 `feature_extraction.text` 模块中提供了英语停用词的内置列表：

In[20]:

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Number of stop words: {}".format(len(ENGLISH_STOP_WORDS)))
print("Every 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

Out[20]:

```
Number of stop words: 318
Every 10th stopword:
['above', 'elsewhere', 'into', 'well', 'rather', 'fifteen', 'had', 'enough',
 'herein', 'should', 'third', 'although', 'more', 'this', 'none', 'seemed',
 'nobody', 'seems', 'he', 'also', 'fill', 'anyone', 'anything', 'me', 'the',
 'yet', 'go', 'seeming', 'front', 'beforehand', 'forty', 'i']
```

显然，删除上述列表中的停用词只能使特征数量减少 318 个（即上述列表的长度），但可能会提高性能。我们来试一下：

In[21]:

```
# 指定stop_words="english"将使用内置列表。
# 我们也可以扩展这个列表并传入我们自己的列表。
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train with stop words:\n{}".format(repr(X_train)))
```

Out[21]:

```
X_train with stop words:
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'
    with 2149958 stored elements in Compressed Sparse Row format>
```

现在数据集中的特征数量减少了 305 个（27271–26966），说明大部分停用词（但不是所有）都出现了。我们再次运行网格搜索：

In[22]:

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

Out[22]:

```
Best cross-validation score: 0.88
```

使用停用词后的网格搜索性能略有下降——不至于担心，但鉴于从 27 000 多个特征中删除 305 个不太可能对性能或可解释性造成很大影响，所以使用这个列表似乎是不值得的。固定的列表主要对小型数据集很有帮助，这些数据集可能没有包含足够的信息，模型从数据本身无法判断出哪些单词是停用词。作为练习，你可以尝试另一种方法，即通过设置 `CountVectorizer` 的 `max_df` 选项来舍弃出现最频繁的单词，并查看它对特征数量和性能有什么影响。

7.5 用tf-idf缩放数据

另一种方法是按照我们预计的特征信息量大小来缩放特征，而不是舍弃那些认为不重要的特征。最常见的一种做法就是使用**词频 – 逆向文档频率**（term frequency–inverse document frequency，tf-idf）方法。这一方法对在某个特定文档中经常出现的术语给予很高的权重，但对在语料库的许多文档中都经常出现的术语给予的权重却不高。如果一个单词在某个特定文档中经常出现，但在许多文档中却不常出现，那么这个单词很可能是对文

档内容的很好描述。scikit-learn 在两个类中实现了 tf-idf 方法：TfidfTransformer 和 TfidfVectorizer，前者接受 CountVectorizer 生成的稀疏矩阵并将其变换，后者接受文本数据并完成词袋特征提取与 tf-idf 变换。tf-idf 缩放方案有几种变体，你可以在维基百科上阅读相关内容（https://en.wikipedia.org/wiki/Tf-idf）。单词 w 在文档 d 中的 tf-idf 分数在 TfidfTransformer 类和 TfidfVectorizer 类中都有实现，其计算公式如下所示：[10]

$$\text{tfidf}(w,d) = \text{tf} \log\left(\frac{N+1}{N_w+1}\right) + 1$$

其中 N 是训练集中的文档数量，N_w 是训练集中出现单词 w 的文档数量，tf（词频）是单词 w 在查询文档 d（你想要变换或编码的文档）中出现的次数。两个类在计算 tf-idf 表示之后都还应用了 L2 范数。换句话说，它们将每个文档的表示缩放到欧几里得范数为 1。利用这种缩放方法，文档长度（单词数量）不会改变向量化表示。

由于 tf-idf 实际上利用了训练数据的统计学属性，所以我们将使用在第 6 章中介绍过的管道，以确保网格搜索的结果有效。这样会得到下列代码：

In[23]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5),
                     LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

Out[23]:

```
Best cross-validation score: 0.89
```

如你所见，使用 tf-idf 代替仅统计词数对性能有所提高。我们还可以查看 tf-idf 找到的最重要的单词。请记住，tf-idf 缩放的目的是找到能够区分文档的单词，但它完全是一种无监督技术。因此，这里的“重要”不一定与我们感兴趣的“正面评论”和“负面评论”标签相关。首先，我们从管道中提取 TfidfVectorizer：

In[24]:

```
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# 变换训练数据集
X_train = vectorizer.transform(text_train)
# 找到数据集中每个特征的最大值
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# 获取特征名称
feature_names = np.array(vectorizer.get_feature_names())

print("Features with lowest tfidf:\n{}".format(
```

注 10：这里给出这个公式主要是为了完整性，你在使用 tf-idf 时无需记住它。

```
    feature_names[sorted_by_tfidf[:20]]))

print("Features with highest tfidf: \n{}".format(
    feature_names[sorted_by_tfidf[-20:]]))
```

Out[24]:

```
Features with lowest tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'
 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker'
 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing' 'downhill'
 'inane']
Features with highest tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'
 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'
 'khouri' 'zizek' 'rob' 'timon' 'titanic']
```

tf-idf 较小的特征要么是在许多文档里都很常用，要么就是很少使用，且仅出现在非常长的文档中。有趣的是，许多 tf-idf 较大的特征实际上对应的是特定的演出或电影。这些术语仅出现在这些特定演出或电影的评论中，但往往在这些评论中多次出现。例如，对于 "pokemon"、"smallville" 和 "doodlebops" 是显而易见的，但这里的 "scanners" 实际上指的也是电影标题。这些单词不太可能有助于我们的情感分类任务（除非有些电影的评价可能普遍偏正面或偏负面），但肯定包含了关于评论的大量具体信息。

我们还可以找到逆向文档频率较低的单词，即出现次数很多，因此被认为不那么重要的单词。训练集的逆向文档频率值被保存在 idf_ 属性中：

In[25]:

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Features with lowest idf:\n{}".format(
    feature_names[sorted_by_idf[:100]]))
```

Out[25]:

```
Features with lowest idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'
 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'
 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'
 'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'
 'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'
 'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'
 'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'
 'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'
 'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'
 'him']
```

正如所料，这些词大多是英语中的停用词，比如 "the" 和 "no"。但有些单词显然是电影评论特有的，比如 "movie"、"film"、"time"、"story" 等。有趣的是，"good"、"great" 和 "bad" 也属于频繁出现的单词，因此根据 tf-idf 度量也属于“不太相关”的单词，尽管我们可能认为这些单词对情感分析任务非常重要。

7.6 研究模型系数

最后，我们详细看一下 Logistic 回归模型从数据中实际学到的内容。由于特征数量非常多(删除出现次数不多的特征之后还有 27 271 个)，所以显然我们不能同时查看所有系数。但是，我们可以查看最大的系数，并查看这些系数对应的单词。我们将使用基于 tf-idf 特征训练的最后一个模型。

下面这张条形图（图 7-2）给出了 Logistic 回归模型中最大的 25 个系数与最小的 25 个系数，其高度表示每个系数的大小：

In[26]:

```
mglearn.tools.visualize_coefficients(
    grid.best_estimator_.named_steps["logisticregression"].coef_,
    feature_names, n_top_features=40)
```

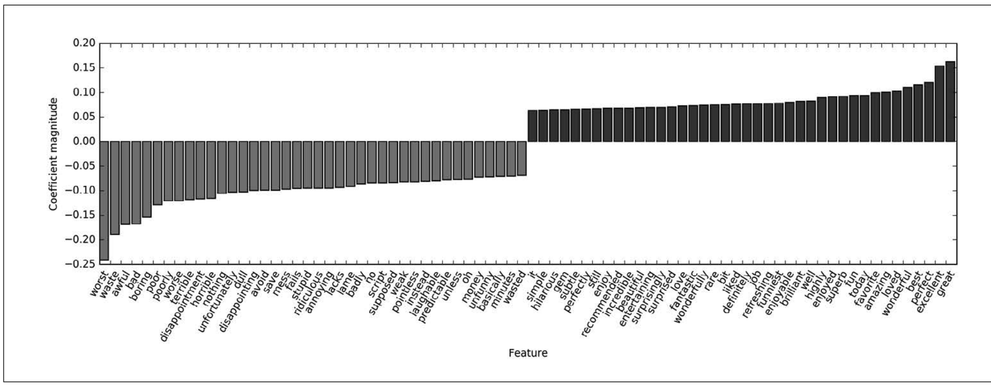


图 7-2：在 tf-idf 特征上训练的 Logistic 回归的最大系数和最小系数

左侧的负系数属于模型找到的表示负面评论的单词，而右侧的正系数属于模型找到的表示正面评论的单词。大多数单词都是非常直观的，比如 "worst"（最差）、"waste"（浪费）、"disappointment"（失望）和 "laughable"（可笑）都表示不好的电影评论，而 "excellent"（优秀）、"wonderful"（精彩）、"enjoyable"（令人愉悦）和 "refreshing"（耳目一新）则表示正面的电影评论。有些词的含义不那么明确，比如 "bit"（一点）、"job"（工作）和 "today"（今天），但它们可能是类似 "good job"（做得不错）和“best today”（今日最佳）等短语的一部分。

7.7 多个单词的词袋（*n*元分词）

使用词袋表示的主要缺点之一是完全舍弃了单词顺序。因此，“it’s bad, not good at all”（电影很差，一点也不好）和“it’s good, not bad at all”（电影很好，还不错）这两个字符串的词袋表示完全相同，尽管它们的含义相反。将“not”（不）放在单词前面，这只是上下文很重要的一个例子（可能是一个极端的例子）。幸运的是，使用词袋表示时有一种获取上下文的方法，就是不仅考虑单一词例的计数，而且还考虑相邻的两个或三个词例的计数。

两个词例被称为**二元分词**（bigram），三个词例被称为**三元分词**（trigram），更一般的词例序列被称为 ***n* 元分词**（*n*-gram）。我们可以通过改变 `CountVectorizer` 或 `TfidfVectorizer` 的 `ngram_range` 参数来改变作为特征的词例范围。`ngram_range` 参数是一个元组，包含要考虑的词例序列的最小长度和最大长度。下面是在之前用过的玩具数据上的一个示例：

In[27]:

```
print("bards_words:\n{}".format(bards_words))
```

Out[27]:

```
bards_words:
['The fool doth think he is wise,',
 'but the wise man knows himself to be a fool']
```

默认情况下，为每个长度最小为 1 且最大为 1 的词例序列（或者换句话说，刚好 1 个词例）创建一个特征——单个词例也被称为**一元分词**（unigram）：

In[28]:

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[28]:

```
Vocabulary size: 13
Vocabulary:
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',
 'think', 'to', 'wise']
```

要想仅查看二元分词（即仅查看由两个相邻词例组成的序列），可以将 `ngram_range` 设置为 `(2, 2)`：

In[29]:

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[29]:

```
Vocabulary size: 14
Vocabulary:
['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to',
 'is wise', 'knows himself', 'man knows', 'the fool', 'the wise',
 'think he', 'to be', 'wise man']
```

使用更长的词例序列通常会得到更多的特征，也会得到更具体的特征。`bard_words` 的两个短语中没有相同的二元分词：

In[30]:

```
print("Transformed data (dense):\n{}".format(cv.transform(bards_words).toarray()))
```

Out[30]:

```
Transformed data (dense):
[[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

对于大多数应用而言，最小的词例数量应该是 1，因为单个单词通常包含丰富的含义。在大多数情况下，添加二元分词会有所帮助。添加更长的序列（一直到五元分词）也可能有所帮助，但这会导致特征数量的大大增加，也可能会导致过拟合，因为其中包含许多非常具体的特征。原则上来说，二元分词的数量是一元分词数量的平方，三元分词的数量是一元分词数量的三次方，从而导致非常大的特征空间。在实践中，更高的 *n* 元分词在数据中的出现次数实际上更少，原因在于（英语）语言的结构，不过这个数字仍然很大。

下面是在 `bards_words` 上使用一元分词、二元分词和三元分词的结果：

In[31]:

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[31]:

```
Vocabulary size: 39
Vocabulary:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think',
 'doth think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is',
 'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise',
 'knows', 'knows himself', 'knows himself to', 'man', 'man knows',
 'man knows himself', 'the', 'the fool', 'the fool doth', 'the wise',
 'the wise man', 'think', 'think he', 'think he is', 'to', 'to be',
 'to be fool', 'wise', 'wise man', 'wise man knows']
```

我们在 IMDb 电影评论数据上尝试使用 `TfidfVectorizer`，并利用网格搜索找出 *n* 元分词的最佳设置：

In[32]:

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# 运行网格搜索需要很长时间，因为网格相对较大，且包含三元分词
param_grid = {"logisticregression__C": [0.001, 0.01, 0.1, 1, 10, 100],
              "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters:\n{}".format(grid.best_params_))
```

Out[32]:

```
Best cross-validation score: 0.91
Best parameters:
{'tfidfvectorizer__ngram_range': (1, 3), 'logisticregression__C': 100}
```

从结果中可以看出，我们添加了二元分词特征与三元分词特征之后，性能提高了一个百分点多一点。我们可以将交叉验证精度作为 `ngram_range` 和 `C` 参数的函数并用热图可视化，正如我们在第 5 章中所做的那样（见图 7-3）：

In[33]:

```
# 从网格搜索中提取分数
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
```

```
# 热图可视化
heatmap = mglearn.tools.heatmap(
    scores, xlabel="C", ylabel="ngram_range", cmap="viridis", fmt="%.3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```

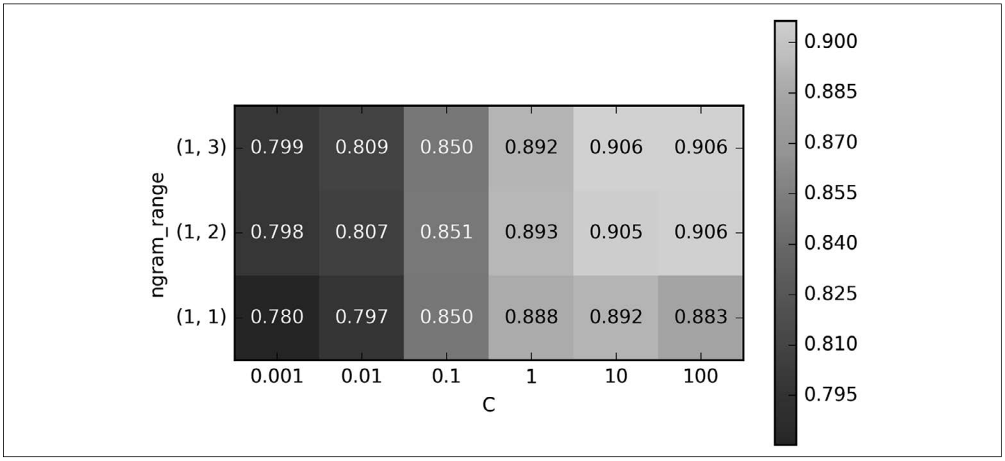


图 7-3：交叉验证平均精度作为参数 ngram_range 和 C 的函数的热图可视化

从热图中可以看出，使用二元分词对性能有很大提高，而添加三元分词对精度只有很小贡献。为了更好地理解模型是如何改进的，我们可以将最佳模型的重要系数可视化，其中包含一元分词、二元分词和三元分词（见图 7-4）：

In[34]:

```
# 提取特征名称与系数
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
```

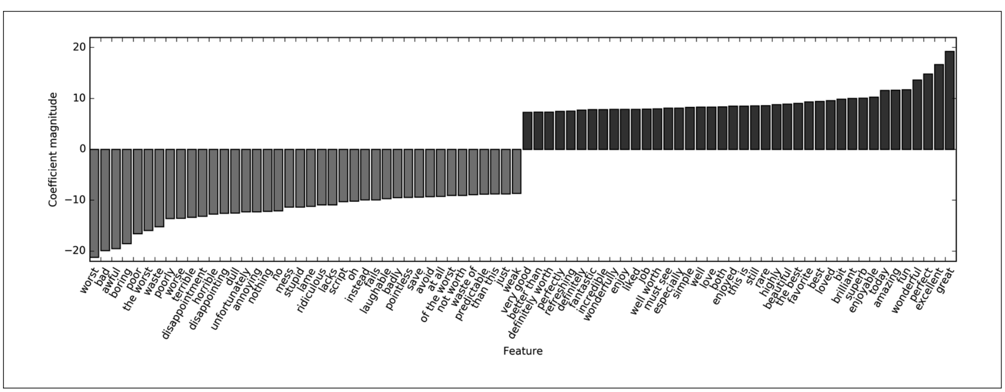


图 7-4：同时使用 tf-idf 缩放与一元分词、二元分词和三元分词时的最重要特征

有几个特别有趣的特征，它们包含单词 "worth"（值得），而这个词本身并没有出现在一元分词模型中："not worth"（不值得）表示负面评论，而 "definitely worth"（绝对值得）和 "well worth"（很值得）表示正面评论。这是上下文影响 "worth" 一词含义的主要示例。

接下来，我们只将三元分词可视化，以进一步深入了解这些特征有用的原因。许多有用的二元分词和三元分词都由常见的单词组成，这些单词本身可能没有什么信息量，比如 "none of the"（没有一个）、"the only good"（唯一好的）、"on and on"（不停地）、"this is one"（这是一部）、"of the most"（最）等短语中的单词。但是，与一元分词特征的重要性相比，这些特征的影响非常有限，正如图 7-5 所示。

In[35]:

```
# 找到三元分词特征
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# 仅将三元分词特征可视化
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                     feature_names[mask], n_top_features=40)
```

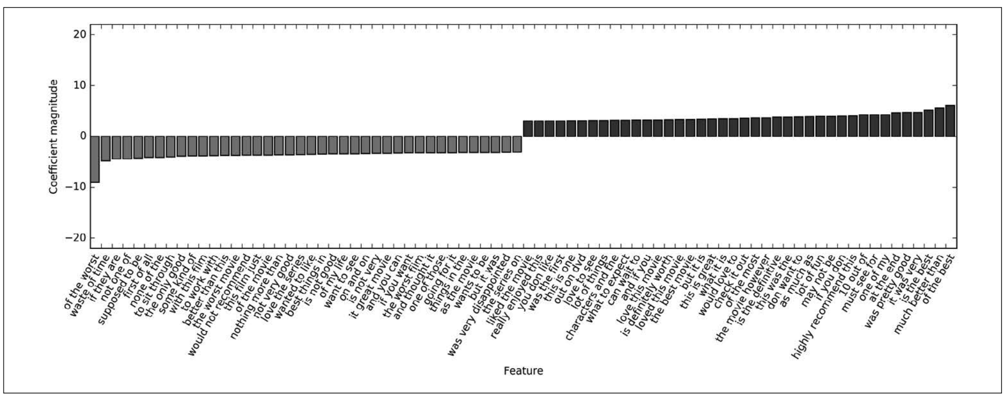


图 7-5：仅将对模型重要的三元分词特征可视化

7.8 高级分词、词干提取与词形还原

如前所述，CountVectorizer 和 TfidfVectorizer 中的特征提取相对简单，还有更为复杂的方法。在更加复杂的文本处理应用中，通常需要改进的步骤是词袋模型的第一步：分词（tokenization）。这一步骤为特征提取定义了一个单词是如何构成的。

我们前面看到，词表中通常同时包含某些单词的单数形式和复数形式，比如 "drawback" 和 "drawbacks"、"drawer" 和 "drawers"、"drawing" 和 "drawings"。对于词袋模型而言，"drawback" 和 "drawbacks" 的语义非常接近，区分二者只会增加过拟合，并导致模型无法充分利用训练数据。同样我们还发现，词表中包含像 "replace"、"replaced"、"replacement"、"replaces" 和 "replacing" 这样的单词，它们都是动词“to replace”的不同动词形式或相关名词。与名词的单复数形式一样，将不同的动词形式及相关单词视为不同的词例，这不利于构建具有良好泛化性能的模型。

这个问题可以通过用**词干**（word stem）表示每个单词来解决，这一方法涉及找出 [或**合并**（conflate）] 所有具有相同词干的单词。如果使用基于规则的启发法来实现（比如删除常见的后缀），那么通常将其称为**词干提取**（stemming）。如果使用的是由已知单词形式组成的字典（明确的且经过人工验证的系统），并且考虑了单词在句子中的作用，那么这个过程被称为**词形还原**（lemmatization），单词的标准化形式被称为**词元**（lemma）。词干提取和词形还原这两种处理方法都是**标准化**（normalization）的形式之一，标准化是指尝试提取一个单词的某种标准形式。标准化的另一个有趣的例子是拼写校正，这种方法在实践中很有用，但超出了本书的范围。

为了更好地理解标准化，我们来对比一种词干提取方法（Porter 词干提取器，一种广泛使用的启发法集合，从 `nltk` 包导入）与 `spacy` 包[11] 中实现的词形还原：[12]

In[36]:

```
import spacy
import nltk

# 加载spacy的英语模型
en_nlp = spacy.load('en')
# 将nltk的Porter词干提取器实例化
stemmer = nltk.stem.PorterStemmer()

# 定义一个函数来对比spacy中的词形还原与nltk中的词干提取
def compare_normalization(doc):
    # 在spacy中对文档进行分词
    doc_spacy = en_nlp(doc)
    # 打印出spacy找到的词元
    print("Lemmatization:")
    print([token.lemma_ for token in doc_spacy])
    # 打印出Porter词干提取器找到的词例
    print("Stemming:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

我们将用一个句子来比较词形还原与 Porter 词干提取器，以显示二者的一些区别：[13]

In[37]:

```
compare_normalization(u"Our meeting today was worse than yesterday, "
                       "I'm scared of meeting the clients tomorrow.")
```

Out[37]:

```
Lemmatization:
['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
 'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
Stemming:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m",
 'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
```

注 11：安装 `spacy` 包之后需要下载相应的语言包，你可以在命令行输入 `python3 -m spacy download en` 来下载英语语言包。——译者注

注 12：想了解接口的细节，请参阅 `nltk`（http://www.nltk.org/）和 `spacy`（https://spacy.io/docs/）的文档。我们这里更关注一般性原则。

注 13：1.7.5 版的 `spacy` 将 `'our'`、`'i'` 等代词全部还原为 `'-PRON-'`，详情请参见 `spacy` 官方文档。——译者注

词干提取总是局限于将单词简化成词干，因此 "was" 变成了 "wa"，而词形还原可以得到正确的动词基本词形 "be"。同样，词形还原可以将 "worse" 标准化为 "bad"，而词干提取得到的是 "wors"。另一个主要区别在于，词干提取将两处 "meeting" 都简化为 "meet"。利用词形还原，第一处 "meeting" 被认为是名词，所以没有变化，而第二处 "meeting" 被认为是动词，所以变为 "meet"。一般来说，词形还原是一个比词干提取更复杂的过程，但用于机器学习的词例标准化时通常可以给出比词干提取更好的结果。

虽然 scikit-learn 没有实现这两种形式的标准化，但 CountVectorizer 允许使用 tokenizer 参数来指定使用你自己的分词器将每个文档转换为词例列表。我们可以使用 spacy 的词形还原了创建一个可调用对象，它接受一个字符串并生成一个词元列表：

In[38]:

```
# 技术细节：我们希望使用由CountVectorizer所使用的基于正则表达式的分词器，
# 并仅使用spacy的词形还原。
# 为此，我们将en_nlp.tokenizer（spacy分词器）替换为基于正则表达式的分词。
import re
# 在CountVectorizer中使用的正则表达式
regexp = re.compile('(?u)\\b\\w\\w+\\b')

# 加载spacy语言模型，并保存旧的分词器
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# 将分词器替换为前面的正则表达式
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(
    regexp.findall(string))

# 用spacy文档处理管道创建一个自定义分词器
# （现在使用我们自己的分词器）
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# 利用自定义分词器来定义一个计数向量器
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

我们变换数据并检查词表的大小：

In[39]:

```
# 利用带词形还原的CountVectorizer对text_train进行变换
X_train_lemma = lemma_vect.fit_transform(text_train)
print("X_train_lemma.shape: {}".format(X_train_lemma.shape))

# 标准的CountVectorizer，以供参考
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train.shape: {}".format(X_train.shape))
```

Out[39]:

```
X_train_lemma.shape:  (25000, 21596)
X_train.shape:  (25000, 27271)
```

从输出中可以看出，词形还原将特征数量从 27 271 个（标准的 CountVectorizer 处理过程）

减少到 21 596 个。词形还原可以被看作是一种正则化，因为它合并了某些特征。因此我们预计，数据集很小时词形还原对性能的提升最大。为了说明词形还原的作用，我们将使用 `StratifiedShuffleSplit` 做交叉验证，仅使用 1% 的数据作为训练数据，其余数据作为测试数据：

In[40]:

```
# 仅使用1%的数据作为训练集来构建网格搜索
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.99,
                            train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
# 利用标准的CountVectorizer进行网格搜索
grid.fit(X_train, y_train)
print("Best cross-validation score "
      "(standard CountVectorizer): {:.3f}".format(grid.best_score_))
# 利用词形还原进行网格搜索
grid.fit(X_train_lemma, y_train)
print("Best cross-validation score "
      "(lemmatization): {:.3f}".format(grid.best_score_))
```

Out[40]:

```
Best cross-validation score (standard CountVectorizer): 0.721
Best cross-validation score (lemmatization): 0.731
```

在这个例子中，词形还原对性能有较小的提高。与许多特征提取技术一样，其结果因数据集的不同而不同。词形还原与词干提取有时有助于构建更好的模型（或至少是更简洁的模型），所以我们建议你，在特定任务中努力提升最后一点性能时可以尝试下这些技术。

7.9　主题建模与文档聚类

常用于文本数据的一种特殊技术是**主题建模**（topic modeling），这是描述将每个文档分配给一个或多个主题的任务（通常是无监督的）的概括性术语。这方面一个很好的例子是新闻数据，它们可以被分为“政治”“体育”“金融”等主题。如果为每个文档分配一个主题，那么这是一个文档聚类任务，正如第 3 章中所述。如果每个文档可以有多个主题，那么这个任务与第 3 章中的分解方法有关。我们学到的每个成分对应于一个主题，文档表示中的成分系数告诉我们这个文档与该主题的相关性强弱。通常来说，人们在谈论主题建模时，他们指的是一种叫作**隐含狄利克雷分布**（Latent Dirichlet Allocation，LDA）的特定分解方法。[14]

隐含狄利克雷分布

从直观上来看，LDA 模型试图找出频繁共同出现的单词群组（即主题）。LDA 还要求，每

注 14：还有另一种机器学习模型通常也简称为 LDA，那就是线性判别分析（Linear Discriminant Analysis），一种线性分类模型。这造成了很多混乱。在本书中，LDA 是指隐含狄利克雷分布。

个文档可以被理解为主题子集的“混合”。重要的是要理解，机器学习模型所谓的“主题”可能不是我们通常在日常对话中所说的主题，而是更类似于 PCA 或 NMF（第 3 章讨论过这些内容）所提取的成分，它可能具有语义，也可能没有。即使 LDA“主题”具有语义，它可能也不是我们通常所说的主题。回到新闻文章的例子，我们可能有许多关于体育、政治和金融的文章，由两位作者所写。在一篇政治文章中，我们预计可能会看到“州长”“投票”“党派”等词语，而在一篇体育文章中，我们预计可能会看到类似“队伍”“得分”和“赛季”之类的词语。这两组词语可能会同时出现，而例如“队伍”和“州长”就不太可能同时出现。但是，这并不是我们预计可能同时出现的唯一的单词群组。这两位记者可能偏爱不同的短语或者选择不同的单词。可能其中一人喜欢使用“划界”（demarcate）这个词，而另一人喜欢使用“两极分化”（polarize）这个词。其他“主题”可能是“记者 A 常用的词语”和“记者 B 常用的词语”，虽然这并不是通常意义上的主题。

我们将 LDA 应用于电影评论数据集，来看一下它在实践中的效果。对于无监督的文本文档模型，通常最好删除非常常见的单词，否则它们可能会支配分析过程。我们将删除至少在 15% 的文档中出现过的单词，并在删除前 15% 之后，将词袋模型限定为最常见的 10 000 个单词：

In[41]:

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

我们将学习一个包含 10 个主题的主题模型，它包含的主题个数很少，我们可以查看所有主题。与 NMF 中的分量类似，主题没有内在的顺序，而改变主题数量将会改变所有主题。[15] 我们将使用 "batch" 学习方法，它比默认方法（"online"）稍慢，但通常会给出更好的结果。我们还将增大 max_iter，这样会得到更好的模型：

In[42]:

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                                max_iter=25, random_state=0)
# 我们在一个步骤中构建模型并变换数据
# 计算变换需要花点时间，二者同时进行可以节省时间
document_topics = lda.fit_transform(X)
```

与第 3 章中所讲的分解方法类似，LatentDirichletAllocation 有一个 components_ 属性，其中保存了每个单词对每个主题的重要性。components_ 的大小为 (n_topics, n_words)：

In[43]:

```
lda.components_.shape
```

Out[43]:

```
(10, 10000)
```

为了更好地理解不同主题的含义，我们将查看每个主题中最重要的单词。print_topics 函数为这些特征提供了良好的格式：

注 15：事实上，NMF 和 LDA 解决的是非常相关的问题，我们也可以用 NMF 来提取主题。

In[44]:

```
# 对于每个主题（components_的一行），将特征排序（升序）
# 用[:, ::-1]将行反转，使排序变为降序
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# 从向量器中获取特征名称
feature_names = np.array(vect.get_feature_names())
```

In[45]:

```
# 打印出前10个主题：
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

Out[45]:

```
topic 0       topic 1       topic 2       topic 3       topic 4
--------      --------      --------      --------      --------
between       war           funny         show          didn
young         world         worst         series        saw
family        us            comedy        episode       am
real          our           thing         tv            thought
performance   american      guy           episodes      years
beautiful     documentary   re            shows         book
work          history       stupid        season        watched
each          new           actually      new           now
both          own           nothing       television    dvd
director      point         want          years         got


topic 5       topic 6       topic 7       topic 8       topic 9
--------      --------      --------      --------      --------
horror        kids          cast          performance   house
action        action        role          role          woman
effects       animation     john          john          gets
budget        game          version       actor         killer
nothing       fun           novel         oscar         girl
original      disney        both          cast          wife
director      children      director      plays         horror
minutes       10            played        jack          young
pretty        kid           performance   joe           goes
doesn         old           mr            performances  around
```

从重要的单词来看，主题 1 似乎是关于历史和战争的电影，主题 2 可能是关于糟糕的喜剧，主题 3 可能是关于电视连续剧，主题 4 可能提取了一些非常常见的单词，而主题 6 似乎是关于儿童电影，主题 8 似乎提取了与获奖相关的评论。仅使用 10 个主题，每个主题都需要非常宽泛，才能共同涵盖我们的数据集中所有不同类型的评论。

接下来，我们将学习另一个模型，这次包含 100 个主题。使用更多的主题，将使得分析过程更加困难，但更可能使主题专门针对于某个有趣的数据子集：

In[46]:

```
lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch",
                                   max_iter=25, random_state=0)
document_topics100 = lda100.fit_transform(X)
```

查看所有 100 个主题可能有点困难，所以我们选取了一些有趣的而且有代表性的主题：

In[47]:

```
topics = np.array([7, 16, 24, 25, 28, 36, 37, 45, 51, 53, 54, 63, 89, 97])

sorting = np.argsort(lda100.components_, axis=1)[:, ::-1]
feature_names = np.array(vect.get_feature_names())
mglearn.tools.print_topics(topics=topics, feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=7, n_words=20)
```

Out[47]:

```
topic 7       topic 16      topic 24      topic 25      topic 28
--------      --------      --------      --------      --------
thriller      worst         german        car           beautiful
suspense      awful         hitler        gets          young
horror        boring        nazi          guy           old
atmosphere    horrible      midnight      around        romantic
mystery       stupid        joe           down          between
house         thing         germany       kill          romance
director      terrible      years         goes          wonderful
quite         script        history       killed        heart
bit           nothing       new           going         feel
de            worse         modesty       house         year
performances  waste         cowboy        away          each
dark          pretty        jewish        head          french
twist         minutes       past          take          sweet
hitchcock     didn          kirk          another       boy
tension       actors        young         getting       loved
interesting   actually      spanish       doesn         girl
mysterious    re            enterprise    now           relationship
murder        supposed      von           night         saw
ending        mean          nazis         right         both
creepy        want          spock         woman         simple


topic 36      topic 37      topic 41      topic 45      topic 51
--------      --------      --------      --------      --------
performance   excellent     war           music         earth
role          highly        american      song          space
actor         amazing       world         songs         planet
cast          wonderful     soldiers      rock          superman
play          truly         military      band          alien
actors        superb        army          soundtrack    world
performances  actors        tarzan        singing       evil
played        brilliant     soldier       voice         humans
supporting    recommend     america       singer        aliens
director      quite         country       sing          human
oscar         performance   americans     musical       creatures
roles         performances  during        roll          miike
actress       perfect       men           fan           monsters
excellent     drama         us            metal         apes
screen        without       government    concert       clark
plays         beautiful     jungle        playing       burton
award         human         vietnam       hear          tim
```

```
work          moving        ii            fans          outer
playing       world         political     prince        men
gives         recommended   against       especially    moon


topic 53      topic 54      topic 63      topic 89      topic 97
--------      --------      --------      --------      --------
scott         money         funny         dead          didn
gary          budget        comedy        zombie        thought
streisand     actors        laugh         gore          wasn
star          low           jokes         zombies       ending
hart          worst         humor         blood         minutes
lundgren      waste         hilarious     horror        got
dolph         10            laughs        flesh         felt
career        give          fun           minutes       part
sabrina       want          re            body          going
role          nothing       funniest      living        seemed
temple        terrible      laughing      eating        bit
phantom       crap          joke          flick         found
judy          must          few           budget        though
melissa       reviews       moments       head          nothing
zorro         imdb          guy           gory          lot
gets          director      unfunny       evil          saw
barbra        thing         times         shot          long
cast          believe       laughed       low           interesting
short         am            comedies      fulci         few
serial        actually      isn           re            half
```

这次我们提取的主题似乎更加具体，不过很多都难以解读。主题 7 似乎是关于恐怖电影和惊悚片，主题 16 和 54 似乎是关于不好的评论，而主题 63 似乎主要是关于喜剧的正面评论。如果想要利用发现的主题做出进一步的推断，那么我们应该查看分配给这些主题的文档，以验证我们通过查看每个主题排名最靠前的单词所得到的直觉。例如，主题 45 似乎是关于音乐的。我们来查看哪些评论被分配给了这个主题：

In[49]:

```
# 按主题45“music”进行排序
music = np.argsort(document_topics100[:, 45])[::-1]
# 打印出这个主题最重要的前5个文档
for i in music[:10]:
    # 显示前两个句子
    print(b".".join(text_train[i].split(b".")[:2]) + b".\n")
```

Out[49]:

```
b'I love this movie and never get tired of watching. The music in it is great.\n'
b"I enjoyed Still Crazy more than any film I have seen in years. A successful
  band from the 70's decide to give it another try.\n"
b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for
  Warner Bros. His directing style had changed or evolved to the point that
  this film does not contain his signature overhead shots or huge production
  numbers with thousands of extras.\n'
b"What happens to washed up rock-n-roll stars in the late 1990's?
  They launch a comeback / reunion tour. At least, that's what the members of
  Strange Fruit, a (fictional) 70's stadium rock group do.\n"
```

```
b'As a big-time Prince fan of the last three to four years, I really can\'t
  believe I\'ve only just got round to watching "Purple Rain". The brand new
  2-disc anniversary Special Edition led me to buy it.\n'
b"This film is worth seeing alone for Jared Harris' outstanding portrayal
  of John Lennon. It doesn't matter that Harris doesn't exactly resemble
  Lennon; his mannerisms, expressions, posture, accent and attitude are
  pure Lennon.\n"
b"The funky, yet strictly second-tier British glam-rock band Strange Fruit
  breaks up at the end of the wild'n'wacky excess-ridden 70's. The individual
  band members go their separate ways and uncomfortably settle into lackluster
  middle age in the dull and uneventful 90's: morose keyboardist Stephen Rea
  winds up penniless and down on his luck, vain, neurotic, pretentious lead
  singer Bill Nighy tries (and fails) to pursue a floundering solo career,
  paranoid drummer Timothy Spall resides in obscurity on a remote farm so he
  can avoid paying a hefty back taxes debt, and surly bass player Jimmy Nail
  installs roofs for a living.\n"
b"I just finished reading a book on Anita Loos' work and the photo in TCM
  Magazine of MacDonald in her angel costume looked great (impressive wings),
  so I thought I'd watch this movie. I'd never heard of the film before, so I
  had no preconceived notions about it whatsoever.\n"
b'I love this movie!!! Purple Rain came out the year I was born and it has had
  my heart since I can remember. Prince is so tight in this movie.\n'
b"This movie is sort of a Carrie meets Heavy Metal. It's about a highschool
  guy who gets picked on alot and he totally gets revenge with the help of a
  Heavy Metal ghost.\n"
```

可以看出，这个主题涵盖许多以音乐为主的评论，从音乐剧到传记电影，再到最后一条评论中难以归类的类型。查看主题还有一种有趣的方法，就是通过对所有评论的 document_topics 进行求和来查看每个主题所获得的整体权重。我们用最常见的两个单词为每个主题命名。图 7-6 给出了学到的主题权重：

In[50]:

```
fig, ax = plt.subplots(1, 2, figsize=(10, 10))
topic_names = ["{:>2} ".format(i) + " ".join(words)
               for i, words in enumerate(feature_names[sorting[:, :2]])]
# 两列的条形图：
for col in [0, 1]:
    start = col * 50
    end = (col + 1) * 50
    ax[col].barh(np.arange(50), np.sum(document_topics100, axis=0)[start:end])
    ax[col].set_yticks(np.arange(50))
    ax[col].set_yticklabels(topic_names[start:end], ha="left", va="top")
    ax[col].invert_yaxis()
    ax[col].set_xlim(0, 2000)
    yax = ax[col].get_yaxis()
    yax.set_tick_params(pad=130)
plt.tight_layout()
```

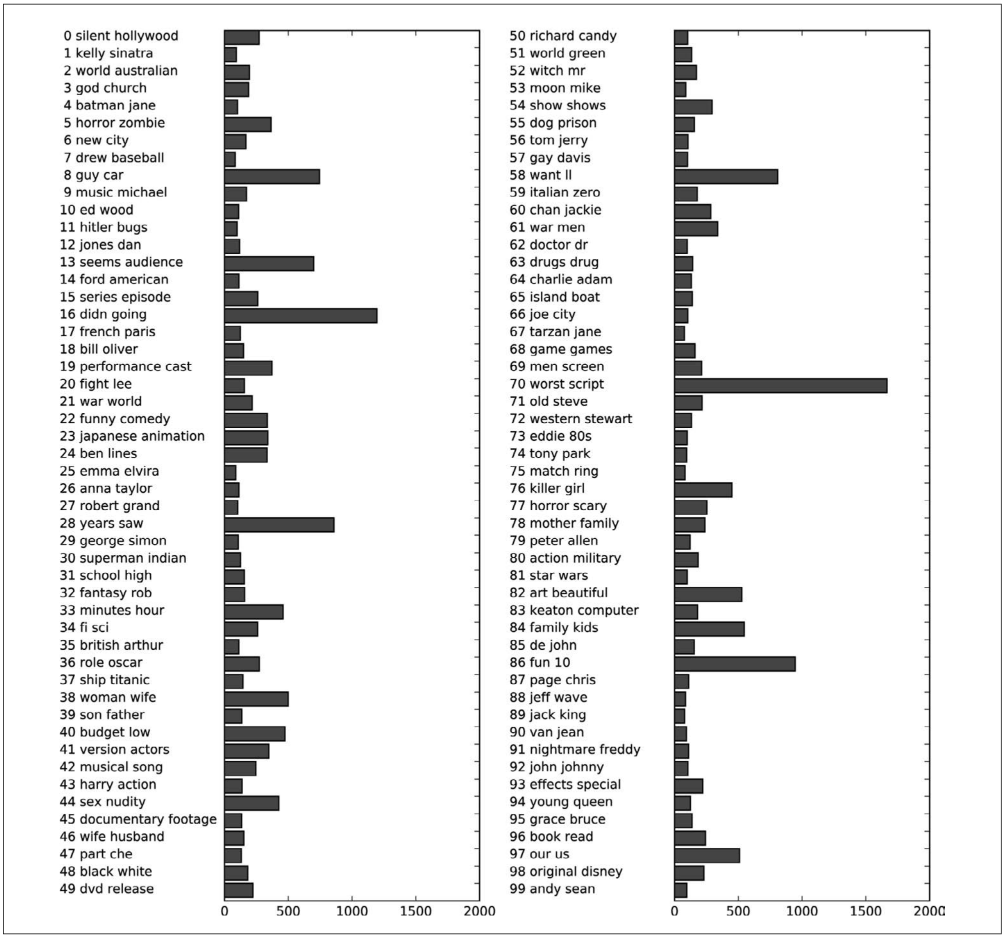


图 7-6：LDA 学到的主题权重

最重要的主题是主题 97，它可能主要包含停用词，可能还有一些稍负面的单词；主题 16 明显是有关负面评论的；然后是一些特定类型的主题与主题 36 和 37，这二者似乎都包含表示赞美的单词。

除了几个不太具体的主题之外，LDA 似乎主要发现了两种主题：特定类型的主题与特定评分的主题。这是一个有趣的发现，因为大部分评论都由一些与电影相关的评论与一些证明或强调评分的评论组成。

在没有标签的情况下（或者像本章的例子这样，即使有标签的情况下），像 LDA 这样的主题模型是理解大型文本语料库的有趣方法。不过 LDA 算法是随机的，改变 `random_state` 参数可能会得到完全不同的结果。虽然找到主题可能很有用，但对于从无监督模型中得出的任何结论都应该持保留态度，我们建议通过查看特定主题中的文档来验证你的直觉。`LDA.transform` 方法生成的主题有时也可以用于监督学习的紧凑表示。当训练样例很少时，这一方法特别有用。

7.10　小结与展望

本章讨论了处理文本 [也叫**自然语言处理**（NLP）] 的基础知识，还给出了一个对电影评论进行分类的示例应用。如果你想要尝试处理文本数据，那么这里讨论的工具应该是很好的出发点。特别是对于文本分类任务，比如检测垃圾邮件和欺诈或者情感分析，词袋模型提供了一种简单而又强大的解决方案。正如机器学习中常见的情况，数据表示是 NLP 应用的关键，检查所提取的词例和 *n* 元分词有助于深入理解建模过程。在文本处理应用中，对于监督任务与无监督任务而言，通常都可以用有意义的方式对模型进行内省，正如我们在本章所见。在实践中使用基于 NLP 的方法时，你应该充分利用这一能力。

自然语言和文本处理是一个很大的研究领域，讨论其高级方法的细节已经远远超出了本书范围。如果你想学习更多内容，我们推荐阅读 Steven Bird、Ewan Klein 和 Edward Loper 合著的 *Natural Language Processing with Python* 一书（O'Reilly 出版社，http://shop.oreilly.com/product/9780596516499.do），其中给出了 NLP 的概述，并介绍了 `nltk` 这个用于 NLP 的 Python 库。另一本很好且概念性更强的书是 Christopher Manning、Prabhakar Raghavan 和 Hinrich Schuütze 合著的标准参考，*Introduction to Information Retrieval*（http://nlp.stanford.edu/IR-book/），其中介绍了信息检索、NLP 和机器学习中的基本算法。这两本书都有可以免费访问的在线版本。如前所述，`CountVectorizer` 类和 `TfidfVectorizer` 类仅实现了相对简单的文本处理方法。对于更高级的文本处理方法，我们推荐使用 Python 包 `spacy`（一个相对较新的包，但非常高效，且设计良好）、`nltk`（一个非常完善且完整的库，但有些过时）和 `gensim`（着重于主题建模的 NLP 包）。

近年来，在文本处理方面有许多非常令人激动的新进展，这些内容都超出了本书的范围，并且都和神经网络有关。第一个进展是使用连续向量表示，也叫作词向量（word vector）或分布式词表示（distributed word representation），它在 `word2vec` 库中实现。Thomas Mikolov（等人的原始论文）"Distributed Representations of Words and Phrases and Their Compo sitionality"（https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf）是对这一主题的很好介绍。对于这篇论文及其后续所讨论的技术，`spacy` 和 `gensim` 都提供了相应的功能。

近年来，NLP 还有另一个研究方向不断升温，就是使用**递归神经网络**（recurrent neural network，RNN）进行文本处理。与只能分配类别标签的分类模型相比，RNN 是一种特别强大的神经网络，可以生成同样是文本的输出。能够生成文本作为输出，使得 RNN 非常适合自动翻译和摘要。Ilya Suskever、Oriol Vinyals 和 Quoc Le 的一篇技术性相对较强的文章"Sequence to Sequence Learning with Neural Networks"（http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf）对这一主题进行了介绍。在 TensorFlow 网站上可以找到使用 `tensorflow` 框架的更为实用的教程（https://www.tensorflow.org/tutorials/seq2seq）。

第 8 章

全书总结

现在你知道了如何使用重要的机器学习算法进行监督学习和无监督学习，这让你可以解决很多种机器学习问题。在带你探索机器学习提供的所有可能性之前，我们希望给你一些最后的建议、一些额外的资源，并提供关于如何进一步提高机器学习和数据科学技能的建议。

8.1 处理机器学习问题

学完了本书介绍的所有强大的方法，你现在可能很想马上行动，开始用你最喜欢的算法来解决数据相关的问题。但这通常并不是开始分析的好方法。机器学习算法通常只是更大的数据分析与决策过程的一小部分。为了有效地利用机器学习，我们需要退后一步，全面地思考问题。首先，你应该思考想要回答什么类型的问题。你想要做探索性分析，只是看看能否在数据中找到有趣的内容？或者你已经有了特定的目标？通常来说，你在开始时有一个目标，比如检测欺诈用户交易、推荐电影或找到未知行星。如果你有这样的目标，那么在构建系统来实现目标之前，你应该首先思考如何定义并衡量成功，以及成功的解决方案对总体业务目标或研究目标有什么影响。假设你的目标是欺诈检测。

然后就出现了下列问题：

- 如何度量欺诈预测是否实际有效？
- 我有没有评估算法的合适数据？
- 如果我成功了，那么我的解决方案会对业务造成什么影响？

正如第 5 章中所述，你最好能使用商业指标直接度量算法的性能，比如增加利润或减少损失。但这通常难以做到。一个更容易回答的问题是："如果我构建出完美的模型，那么会怎么样？"如果完美地检测出所有欺诈行为可以为你的公司每月节省 100 美元，那么这些

省下来的钱可能都不够保证让你开始动手开发算法。如果模型可能为你的公司每月节省上万美元，那么这个问题就值得探索。

假设你已经定义好了要解决的问题，知道一种解决方案可能对你的项目产生重大影响；此外，你还确信拥有合适的信息来评估模型是否成功。接下来的步骤通常是获取数据并构建工作原型。本书中我们讨论过许多你可以使用的模型，以及如何正确地评估和调节这些模型。但在尝试这些模型时请记住，这只是更大的数据科学工作流程中的一小部分，模型构建通常是“收集新数据、清洗数据、构建模型和分析模型”这个反馈环路的一部分。分析模型所犯的错误通常告诉我们：数据中缺失了哪些内容、还可以收集哪些额外数据，或者如何重新规划任务使机器学习更加高效。收集更多数据或不同的数据，或者稍微改变任务规划，可能会比连续运行网格搜索来调参的回报更高。

参与决策过程的人

你还应该考虑是否应该让人参与决策过程，以及如何参与。有些过程（比如无人驾驶汽车的行人检测）需要立即做出决定。其他过程可能不需要立刻的响应，所以可以让人来决定不确定的决策。举个例子，医疗应用可能需要非常高的精度，单靠机器学习算法可能无法达到。但如果一个算法可以自动完成 90%、50% 甚至只有 10% 的决策过程，那么都可能已经减少了响应时间或降低了成本。许多应用都以“简单情况”为主，算法可以对其做出决策，还有相对较少的“复杂情况”，可以将其重新交给人来决定。

8.2　从原型到生产

本书中讨论的工具对许多机器学习应用来说都是很好的，可以非常快速地进行分析和原型设计。许多组织，甚至是非常大型的组织，比如国际银行和全球社交媒体公司，也将 Python 和 `scikit-learn` 用于生产系统。但是，许多公司拥有复杂的基础架构，将 Python 集成到这些系统中并不总是很容易。这不一定是个问题。在许多公司中，数据分析团队使用 Python 或 R 等语言，可以对想法进行快速测试，而生产团队则使用 Go、Scala、C++ 和 Java 等语言来构建鲁棒性更好的可扩展系统。数据分析的需求与构建实时服务并不相同，所以这些任务使用不同的语言是有道理的。一个相对常见的解决方案是使用一种高性能语言在更大的框架内重新实现分析团队找到的解决方案。这种方法比嵌入整个库或整个编程语言并与不同数据格式互相转换要更加简单。

无论你能否在生产系统中使用 `scikit-learn`，重要的是要记住，生产系统的要求与一次性的分析脚本不同。如果将一个算法部署到更大的系统中，那么会涉及软件工程方面的很多内容，比如可靠性、可预测性、运行时间和内存需求。对于在这些领域表现良好的机器学习系统来说，简单就是关键。请仔细检查数据处理和预测流程中的每一部分，并问你自己这些问题：每个步骤增加了多少复杂度？每个组件对数据或计算基础架构的变化的鲁棒性有多高？每个组件的优点能否使其复杂度变得合理？如果你正在构建复杂的机器学习系统，我们强烈推荐阅读 Google 机器学习团队的研究者发布的这篇论文：“Machine Learning: The High Interest Credit Card of Technical Debt”（http://research.google.com/pubs/pub43146.html）。这篇文章重点介绍了在大规模生产中创建并维护机器学习软件的权衡。

虽然技术债问题在大规模的长期项目中特别紧迫，但即使是短期和较小的系统，吸取的教训也有助于我们构建更好的软件。

8.3 测试生产系统

在这本书中，我们介绍了如何基于事先收集的测试集来评估算法的预测结果。这被称为**离线评估**（offline evaluation）。但如果你的机器学习系统是面向用户的，那么这只是评估算法的第一步。下一步通常是**在线测试**（online testing）或**实时测试**（live testing），对在整个系统中使用算法的结果进行评估。改变网站向用户呈现的推荐结果或搜索结果，可能会极大地改变用户行为，并导致意想不到的结果。为了防止出现这种意外，大部分面向用户的服务都会采用 **A/B 测试**（A/B testing），这是一种盲的（blind）用户研究形式。在 A/B 测试中，在用户不知情的情况下，为选中的一部分用户提供使用算法 A 的网站或服务，而为其余用户提供算法 B。对于两组用户，在一段时间内记录相关的成功指标。然后对算法 A 和算法 B 的指标进行对比，并根据这些指标在两种方法中做出选择。使用 A/B 测试让我们能够在实际情况下评估算法，这可能有助于我们发现用户与模型交互时的意外后果。通常情况下，A 是一个新模型，而 B 是已建立的系统。在线测试中还有比 A/B 测试更为复杂的机制，比如 **bandit 算法**。John Myles White 的 *Bandit Algorithms for Website Optimization*（O'Reilly 出版社，http://shop.oreilly.com/product/0636920027393.do）一书对这一主题做出了很好的介绍。

8.4 构建你自己的估计器

本书包含 scikit-learn 中实现的大量工具和算法，可用于各种类型的任务。但是，你通常需要对数据做一些特殊处理，这些处理方法没有在 scikit-learn 中实现。在将数据传入 scikit-learn 模型或管道之前，只做数据预处理可能也足够了。但如果你的预处理依赖于数据，而且你还想使用网格搜索或交叉验证，那么事情就变得有点复杂了。

我们在第 6 章中讨论过将所有依赖于数据的处理过程放在交叉验证循环中的重要性。那么如何同时使用你自己的处理过程与 scikit-learn 工具？有一种简单的解决方案：构建你自己的估计器！实现一个与 scikit-learn 接口兼容的估计器是非常简单的，从而可以与 Pipeline、GridSearchCV 和 cross_val_score 一起使用。你可以在 scikit-learn 文档中找到详细说明（http://scikit-learn.org/stable/developers/contributing.html#rolling-your-own-estimator），但下面是其要点。实现一个变换器类的最简单的方法，就是从 BaseEstimator 和 TransformerMixin 继承，然后实现 `__init__`、fit 和 predict 函数，如下所示。

In[1]:
```
from sklearn.base import BaseEstimator, TransformerMixin

class MyTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, first_parameter=1, second_parameter=2):
        # 所有参数必须在__init__函数中指定
        self.first_parameter = 1
        self.second_parameter = 2
```

```
    def fit(self, X, y=None):
        # fit应该只接受X和y作为参数
        # 即使你的模型是无监督的，你也需要接受一个y参数！

        # 下面是模型拟合的代码
        print("fitting the model right here")
        # fit返回self
        return self

    def transform(self, X):
        # transform只接受X作为参数

        # 对X应用某种变换
        X_transformed = X + 1
        return X_transformed
```

实现一个分类器或回归器的方法是类似的，你只需要从 `ClassifierMixin` 或 `RegressorMixin` 继承，而不是 `TransformerMixin`。此外，你还要实现 `predict`，而不必实现 `transform`。

从上面的例子中可以看出，实现你自己的估计器需要很少的代码，随着时间的推移，大部分 `scikit-learn` 用户都会构建出一组自定义模型。

8.5 下一步怎么走

本书对机器学习进行了介绍，并让你成为一名高效的从业者。但是，如果你想要进一步提高机器学习技能，下面是一些关于书籍和更专业的资源的建议，以便你进一步深入研究。

8.5.1 理论

在本书中，我们试图直观地解释大多数常见机器学习算法的工作原理，而不要求你在数学或计算机科学方面具有坚实的基础。但是，我们讨论的许多模型都使用了概率论、线性代数和最优化方面的理论。虽然没有必要理解这些算法的所有实现细节，但我们认为，了解算法背后的一些理论知识可以让你成为更优秀的数据科学家。关于机器学习理论有许多好书，如果我们所讲的内容激起了你对机器学习可能性的兴趣，那么我们建议你挑选至少一本书深入阅读。我们在前言中已经提到过 Hastie、Tibshirani 和 Friedman 合著的《统计学习基础》一书，这里值得重新推荐一下。另一本相当好读的书是 Stephen Marsland 的 *Machine Learning: An Algorithmic Perspective*（Chapman and Hall/CRC 出版社），它还附带有 Python 代码。还有两本强烈推荐的经典著作：一本是 Christopher Bishop 的 *Pattern Recognition and Machine Learning*（Springer 出版社），着重于概率框架；另一本是 Kevin Murphy 的 *Machine Learning: A Probabilistic Perspective*（MIT 出版社），全面论述了机器学习方法（1000 多页），深入介绍了最先进的方法，内容比本书要丰富得多。

8.5.2 其他机器学习框架和包

虽然 `scikit-learn` 是我们最喜欢的机器学习软件包 [1]，Python 也是我们最喜欢的机器学习语

注 1：Andreas 在这件事上可能并不完全客观。

言，但还有许多其他选择。根据你的需求，Python 和 `scikit-learn` 可能不是你在特定情况下的最佳选择。通常情况下，Python 很适合尝试与评估模型，但更大型的 Web 服务和应用更常用 Java 或 C++ 编写，部署模型可能需要与这些系统进行集成。你想要考虑 `scikit-learn` 之外的选择可能还有一个原因，就是你对统计建模和推断比对预测更感兴趣。在这种情况下，你应该考虑使用 Python 的 `statsmodels` 包，它用更具有统计学意义的接口实现了多种线性模型。如果你还没有专情于 Python，那么还可以考虑使用 R，这是数据科学家的另一种语言。R 是专为统计分析设计的语言，因其出色的可视化功能和许多可用的统计建模包（通常是非常专业化的）而闻名。

另一个常用的机器学习软件包是 `vowpal wabbit`（通常简称为 `vw`，以避免绕口），一个用 C++ 编写的高度优化的机器学习包，还有命令行界面。`vw` 对大型数据集和流数据特别有用。对于在集群上分布式运行的机器学习算法，在写作本书时最常用的解决方案之一是 `mllib`，一个基于 `spark` 分布式计算环境构建的 Scala 库。

8.5.3　排序、推荐系统与其他学习类型

本书是一本入门书，所以我们重点介绍最常见的机器学习任务：监督学习中的分类与回归，无监督学习中的聚类和信号分解。还有许多类型的机器学习，都有很多重要的应用。有两个特别重要的主题没有包含在本书中。第一个是**排序问题**（ranking），对于特定查询，我们希望检索出按相关性排序的答案。你今天可能已经使用过排序系统，它是搜索引擎的运行原理。你输入搜索查询并获取答案的有序列表，它们按相关性进行排序。Manning、Raghavan 和 Schuütze 合著的 *Introduction to Information Retrieval* 一书给出了对排序问题的很好介绍。第二个主题是**推荐系统**（recommender system），就是根据用户偏好向他们提供建议。你可能已经在“您可能认识的人”“购买此商品的顾客还购买了”或“您的最佳选择”等标题下遇到过推荐系统。关于这一主题有大量文献，如果想立刻投身于这一主题，你可能对目前经典的“Netflix 大奖挑战”（Netflix prize challenge，http://www.netflixprize.com/）感兴趣。Netflix 视频流网站发布了关于电影偏好的大型数据集，并对给出最佳推荐的团队奖励一百万美元。另一种常见的应用是时间序列预测（比如股票价格），这方面也有大量的文献。还有许多类型的机器学习任务，比我们这里列出的要多得多，我们建议你从书籍、研究论文和在线社区中获取信息，以找到最适合你实际情况的范式。

8.5.4　概率建模、推断与概率编程

大部分机器学习软件包都提供了预定义的机器学习模型，每种模型应用了一种特定算法。但是，许多现实世界的问题都具有特殊的结构，如果将这种结构正确地纳入模型，则可以得到性能更好的预测。通常来说，具体问题的结构可以用概率论的语言来表述。这种结构通常来自于你想要预测的情况的数学模型。为了理解结构化问题的含义，请思考下面这个例子。

假设你想要构建一个在户外空间提供非常详细的位置估计的移动应用，以帮助用户定位历史遗迹。手机提供了许多传感器来帮你获取精确的位置测量，比如 GPS、加速度计和指南针。你可能还有该区域的精确地图。这个问题是高度结构化的。你从地图中知道了感兴趣地点的位置和路径。你还从 GPS 中得到了粗略的位置，而用户设备中的加速度计和指南针

可以为你提供非常精确的相对测量。但是，将所有这些工具全部放入黑盒机器学习系统中来预测位置，可能并不是最好的主意。这将会丢掉你已经知道的关于现实世界如何运行的所有信息。如果指南针和加速度计告诉你一名用户正在向北走，而 GPS 告诉你该用户正在向南走，你可能不相信 GPS。如果位置估计告诉你用户刚刚走过一堵墙，你应该也会非常怀疑。可以使用概率模型来表述这种情况，然后再使用机器学习或概率推断来找出你应该对每种测量方法的信任程度，并推断出该用户位置的最佳猜测。

一旦你用正确的方式对现状和不同因素共同作用的模型进行表述，那么就有一些方法可以利用这些自定义模型直接计算出预测结果。这些方法中最普遍的方法被称为概率编程语言，它们提供了一种非常优雅又非常紧凑的方法来表述学习问题。概率编程语言的常见例子有 `PyMC`（可用于 Python）和 `Stan`（可用于多种语言的框架，包括 Python）。虽然这些软件包需要你对概率论有一些了解，但它们大大简化了新模型的创建过程。

8.5.5 神经网络

虽然我们在第 2 章和第 7 章都简要涉及了神经网络的主题，但这是机器学习快速发展的领域，每周都会发布新方法和新应用。机器学习和人工智能领域的最新突破都由这些进步所驱动，比如 Alpha Go 程序在围棋比赛中战胜人类冠军、语音理解的性能不断提高，以及接近实时的语音翻译的出现。虽然这一领域的进步非常迅速，以致当前对最新进展的任何参考很快都会过时，但 Ian Goodfellow、Yoshua Bengio 和 Aaron Courville 的新书 *Deep Learning*（MIT 出版社）对这一主题进行了全面介绍。[2]

8.5.6 推广到更大的数据集

在本书中我们总是假设，所处理的数据可以被存储为内存（RAM）中的一个 NumPy 数组或 SciPy 稀疏矩阵。即使现代服务器通常都具有数百 GB 的 RAM，但这也是对你所能处理数据大小的根本限制。不是所有人都买得起这么大型的机器，甚至连从云端供应商租一台都负担不起。不过在大多数应用中，用于构建机器学习系统的数据量相对较小，很少有机器学习数据集包含数百 GB 以上的数据。在多数情况下，这让扩展内存或从云端供应商租一台机器变成可行的解决方案。但是，如果你需要处理 TB 级别的数据，或者需要节省处理大量数据的费用，那么有两种基本策略：**核外学习**（out-of-core learning）与**集群上的并行化**（parallelization over a cluster）。

核外学习是指从无法保存到主存储器的数据中进行学习，但在单台计算机上（甚至是一台计算机的单个处理器）进行学习。数据从硬盘或网络等来源进行读取，一次读取一个样本或者多个样本组成的数据块，这样每个数据块都可以读入 RAM。然后处理这个数据子集并更新模型，以体现从数据中学到的内容。然后舍弃该数据块，并读取下一块数据。`scikit-learn` 中的一些模型实现了核外学习，你可以在在线用户指南中找到相关细节（http://scikit-learn.org/stable/modules/scaling_strategies.html#scaling-with-instances-using-out-of-core-learning）。因为核外学习要求所有数据都由一台计算机处理，所以在非常大的数据集上的运行时间可能很长。此外，并非所有机器学习算法都可以用这种方法实现。

注 2：*Deep Learning* 的预印本可在 http://www.deeplearningbook.org/ 查看。

另一种扩展策略是将数据分配给计算机集群中的多台计算机，让每台计算机处理部分数据。对于某些模型来说这种方法要快得多，而可以处理的数据大小仅受限于集群大小。但是，这种计算通常需要相对复杂的基础架构。目前最常用的分布式计算平台之一是在 Hadoop 之上构建的 spark 平台。spark 在 MLLib 包中包含一些机器学习功能。如果你的数据已经位于 Hadoop 文件系统中，或者你已经使用 spark 来预处理数据，那么这可能是最简单的选项。但如果你还没有这样的基础架构，建立并集成一个 spark 集群可能花费过大。前面提到的 vw 包提供了一些分布式功能，在这种情况下可能是更好的解决方案。

8.5.7　磨练你的技术

与生活中的许多事情一样，只有实践才能让你成为本书所介绍主题方面的专家。对于不同的任务和不同的数据集，特征提取、预处理、可视化和模型构建可能差异很大。你或许足够幸运，已经能够访问大量数据集和任务。如果你还没有想到什么任务，那么一个好的起点是机器学习竞赛，它会发布一个数据集和一个给定任务，许多团队为得到最佳预测结果而展开竞争。许多公司、非盈利组织和大学都会举办这种比赛。要想找到这些比赛，最常去的地方之一是 Kaggle（https://www.kaggle.com/），这是一个定期举办数据科学比赛的网站，其中一些比赛会提供大量奖金。

Kaggle 论坛也是关于机器学习最新工具和技巧的很好的信息来源，在网站上可以找到大量数据集。在 OpenML 平台（http://www.openml.org/）上可以找到更多的数据集及相关任务，该平台拥有 20 000 多个数据集，以及 50 000 多个相关的机器学习任务。处理这些数据集可以提供练习机器学习技能的好机会。比赛的一个缺点是，提供了特定的指标来优化，通常还提供了一个固定的、已经预处理过的数据集。请记住，定义问题和收集数据也是现实世界问题的重要方面，用正确的方式表示问题可能比努力提高分类器精度的最后一个百分点要重要得多。

8.6　总结

我们希望已经让你相信了机器学习在大量应用中的实用性，以及在实践中实现机器学习的简单性。继续挖掘数据，同时不要忽视大局。

关于作者

Andreas Müller 在波恩大学获得了机器学习博士学位。他曾是亚马逊公司计算机视觉应用的机器学习研究员，一年后加入了纽约大学的数据科学中心。在过去的四年里，他一直是 scikit-learn 的维护者，也一直是 scikit-learn 的核心贡献者之一。scikit-learn 是工业界和学术界广泛使用的一个机器学习工具箱。Andreas 还创建了其他一些广泛使用的机器学习软件包，或为之做出了贡献。他的任务是创建开源工具，以降低机器学习应用的入门门槛、推动可重复的科学，以及普及高质量的机器学习算法。

Sarah Guido 是一名数据科学家，在创业公司中工作了很长时间。她热爱 Python、机器学习、海量数据和技术世界。作为一名出色的会议演讲者，Sarah 进入了密歇根大学研究生院，目前居住在纽约市。

关于封面

这本书封面上的动物是一只美洲大鲵（hellbender salamander，拉丁名为 Cryptobranchus alleganiensis），是一种原产于美国东部（从纽约到佐治亚）的两栖动物。它有许多有趣的绰号，包括“阿勒格尼鳄鱼”“鼻涕水獭”和“泥中恶魔”。“hellbender”（字面意思是“地狱狂欢”）这个名字的起源已不可考：一种理论认为，早期移民认为美洲大鲵的样子令人不安，并认为它是设法返回地狱的恶魔生物。

美洲大鲵是隐鳃鲵科的成员，最长可达 74 厘米。它是世界上第三大的水生大鲵。它的身体相当扁平，体侧有很厚的皮肤褶皱。虽然在脖子的两侧各有一个腮，但美洲大鲵主要靠皮肤褶皱进行呼吸：气体通过靠近皮肤表面的毛细血管进出体内。

因此，它们的理想栖息地是清澈、流速快、浅水的溪流，可以为它们提供充足的氧气。美洲大鲵躲在岩石下面，主要靠嗅觉捕食，不过它也能探测到水中的振动。它的食物包括小龙虾、小鱼，偶尔也吃同类的卵。美洲大鲵还是其生态系统中一种重要的猎物，其捕食者包括各种鱼、蛇和龟。

美洲大鲵的数量在过去几十年里急剧减少。水质是最大的问题，因为它们的呼吸系统使其对被污染的水或浑水非常敏感。它们栖息地附近的农业活动和其他人类活动越来越多，导致水中的沉积物和化学品越来越多。为了拯救这种濒危物种，生物学家已经开始人工养殖两栖动物，并在它们成长到不那么脆弱之后将其放生。

O’Reilly 图书封面上的很多动物都是濒危动物，它们对世界都很重要。想要详细了解如何帮助这些动物，请访问 animals.oreilly.com。

封面图片来自于 Wood 的 *Animate Creation*。

Python机器学习基础教程

机器学习已成为许多商业应用和研究项目不可或缺的一部分，海量数据使得机器学习的应用范围远超人们想象。本书将向所有对机器学习技术感兴趣的初学者展示，自己动手构建机器学习解决方案并非难事！

书中重点讨论机器学习算法的实践而不是背后的数学，全面涵盖在实践中实现机器学习算法的所有重要内容，帮助读者使用Python和scikit-learn库一步一步构建一个有效的机器学习应用。

- 机器学习的基本概念及其应用
- 常用机器学习算法的优缺点
- 机器学习所处理的数据的表示方法，包括重点关注数据的哪些方面
- 模型评估和调参的高级方法
- 管道的概念
- 处理文本数据的方法，包括文本特有的处理方法
- 进一步提高机器学习和数据科学技能的建议

"对于想用Python进行机器学习的人来说，这本书是超级实用的学习资料。要是我刚开始用scikit-learn时就有这本书该多好啊！"

——Hanna Wallach
微软研究院高级研究员

Andreas C. Müller，scikit-learn库维护者和核心贡献者。现任哥伦比亚大学数据科学研究院讲师，曾任纽约大学数据科学中心助理研究员、亚马逊公司计算机视觉应用的机器学习研究员。在波恩大学获得机器学习博士学位。

Sarah Guido，Mashable公司数据科学家，曾担任Bitly公司首席数据科学家。

PYTHON/MACHINE LEARNING

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／机器学习

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-47561-9

ISBN 978-7-115-47561-9
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报

微博 @图灵社区：电子书和好文章的消息

微博 @图灵新知：图灵教育的科普小组

微信 图灵访谈：ituring_interview，讲述码农精彩人生

微信 图灵教育：turingbooks